Red Hat
Training and Certification

Student Workbook

Red Hat OpenShift Container Platform 4.6 DO101X

Advanced Application Management with Red Hat OpenShift

Edition 2

DO101X-RHOSCP4.6-en-2-7a6f4c2

Red Hat
Learning Community

Join a community dedicated to learning open source

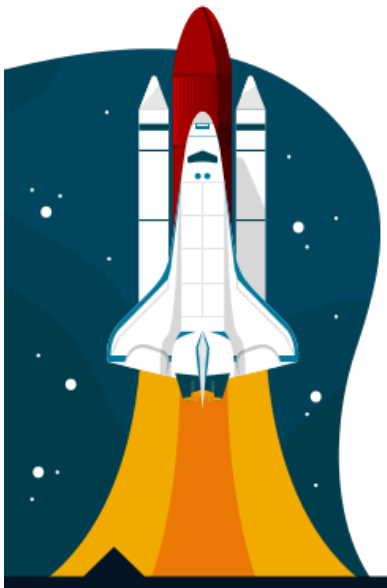
The Red Hat® Learning Community is a collaborative platform for users to accelerate open source skill adoption while working with Red Hat products and experts.

Network with tens of thousands of comunity members

Engage in thousands of active conversations and posts

Join and interact with hundreds of certified training instructors

Unlock badges as you participate and accomplish new goals



This knowledge-sharing platform creates a space where learners can connect, ask questions, and collaborate with other open source practitioners.

Access free Red Hat training videos

Discover the latest Red Hat Training and Certification news

Connect with your instructor - and your classmates - before, after, and during your training course.

Join peers as you explore Red Hat products

Join the conversation learn.redhat.com

Red Hat

Advanced Application Management with Red Hat OpenShift

Red Hat OpenShift Container Platform 4.6 DO101X
Advanced Application Management with Red Hat OpenShift
Edition 2 7a6f4c2
Publication date 20210000

Authors:	Enol Álvarez de Prado, Guy Bianco IV, Marek Czernek, Zach Gutterman, Dan Kolepp, Herve Quatremain, Eduardo Ramírez Ronco, Randy Thomas
Course Architect:	Ravishankar Srinivasan
Editors:	Sam Ffrench, Nicole Muller

Contributors: Richard Allred, Joel Birchler, Nicolette Lucas, David Sacco, Heider Souza, Sajith Sugathan, Chris Tusa

Document Conventions

This section describes various conventions and practices used throughout all Red Hat Training courses.

Admonitions

Red Hat Training courses use the following admonitions:

References
These describe where to find external documentation relevant to a subject.

Note
These are tips, shortcuts, or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on something that makes your life easier.

Important
These provide details of information that is easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring these admonitions will not cause data loss, but may cause irritation and frustration.

Warning
These should not be ignored. Ignoring these admonitions will most likely cause data loss.

Inclusive Language

Red Hat Training is currently reviewing its use of language in various areas to help remove any potentially offensive terms. This is an ongoing process and requires alignment with the products and services covered in Red Hat Training courses. Red Hat appreciates your patience during this process.

Introduction

Advanced Application Management with Red Hat OpenShift

Red Hat OpenShift Container Platform is a containerized application platform that allows enterprises to accelerate and streamline application development, delivery, and deployment on-premise or in the cloud. As OpenShift and Kubernetes continue to be widely adopted by enterprises, developers are increasingly required to understand how to develop, build, and deploy applications with a containerized application platform. While some developers are interested in managing the underlying OpenShift infrastructure, most developers want to focus their time and energy on developing applications, and leveraging OpenShift for its simple building, deployment, and scaling capabilities.

Advanced Application Management with Red Hat OpenShift (DO101x) introduces developers to Red Hat OpenShift Container Platform.

Orientation to the Classroom Environment

DO101x is a *Bring Your Developer Workstation (BYDW)* class, where you use your own internet-enabled system to access the shared OpenShift cluster. The following operating systems are supported:

- Red Hat Enterprise Linux 8 or Fedora Workstation 32 or later
- Ubuntu 20.04 LTS or later
- Microsoft Windows 10
- macOS 10.15 or later

BYDW System Requirements

Attribute	*Minimum Requirements*	*Recommended*
CPU	1.6 GHz or faster processor	Multi-core i7 or equivalient
Memory	8 GB	16 GB or more
Disk	10 GB free space HD	10 GB or more free space SSD
Display Resolution	1024x768	1920x1080 or greater

You must have permissions to install additional software on your system. Some hands-on learning activities in DO101x provide instructions to install the following programs:

- Node.js
- Git 2.18 or later (Git Bash for Windows systems)
- The OpenShift CLI (`oc`) 4.6.0 or later

You might already have these tools installed. If you do not, then wait until the day you start this course to ensure a consistent course experience.

BYDW Systems Support Considerations

Depending on your system, you might see differences between your command-line shell and the examples given in this course.

Red Hat Enterprise Linux or Fedora Workstation

- If you use Bash as the default shell, then your prompt might match the `[user@host ~]$` prompt used in the course examples, although different Bash configurations can produce different results.
- If you use another shell, such as `zsh`, then your prompt format will differ from the prompt used in the course examples.

- When performing the exercises, interpret the `[user@host ~]$` prompt used in the course as a representation of your system prompt.

Ubuntu

- You might find differences in the prompt format.
- In Ubuntu, your prompt might be similar to `user@host:~$`.
- When performing the exercises, interpret the `[user@host ~]$` prompt used in the course as a representation of your Ubuntu prompt.

macOS

- You might find differences in the prompt format.
- In macOS, your prompt might be similar to `host:~ user$`.
- When performing the exercises, interpret the `[user@host ~]$` prompt used in the course as a representation of your macOS prompt.
- You might need to grant execution permissions to the installed runtimes.

Microsoft Windows

- Windows does not support Bash natively. Instead, you must use PowerShell.
- In Windows PowerShell, your prompt should be similar to `PS C:\Users\user>`.
- When performing the exercises, interpret the `[user@host ~]$` Bash prompt as a representation of your Windows PowerShell prompt.
- For some commands, Bash syntax and PowerShell syntax are similar, such as `cd` or `ls`. You can also use the slash character (`/`) in file system paths.
- This course only provides support for Windows PowerShell.
- The Windows firewall might ask for additional permissions in certain exercises.

Executing Long Commands

This course breaks long commands into multiple lines by using the backslash character (`\`), for example:

```
[user@host ~]$ long command \
  argument_1 \
  argument_2
```

The preceding example works on the Linux and macOS systems.

On Windows, use the backtick character (`` ` ``), for example:

```
[user@host ~]$ long command `
  argument_1 `
  argument_2
```

Alternatively, you can type commands in one line on all systems, such as:

```
[user@host ~]$ long command argument_1 argument_2
```

Additional Accounts and Services

Additional Accounts and Services

This course further requires the following accounts:

- A personal GitHub account.
- A Red Hat account.

This course does not require paid versions of any of the preceding accounts.

Chapter 1

Configuring a Cloud Application Developer Environment

Goal

Configure a developer environment with a modern integrated developer environment and version control.

Objectives

- Edit application source code with VSCode.
- Create a Git repository.
- Use version control to collaborate and manage application source code.

Sections

- Developing Applications with VSCode (and Guided Exercise)
- Initializing a Git Repository (and Guided Exercise)
- Managing Application Source Code with Git (and Guided Exercise)

Developing Applications with VSCode

Objectives

After completing this section, you should be able to edit application source code with VSCode.

Integrated Development Environments

Software developers execute many different types of tasks during the software development life cycle of an application, such as:

- Compile and build the source code
- Correct syntax errors
- Debug runtime errors
- Maintain version control changes to the source code
- Refactor source code
- Execute source code tests

In the past, developers used a collection of separate tools, such as editors, compilers, interpreters, and debuggers, to develop software applications. Inefficiencies arose from using separate tools, leading to the creation of **Integrated Development Environments**, or IDEs. IDEs improve developer productivity by integrating common software development tools into a single application. IDEs often integrate the following:

- Language-specific editors
- Code completion capabilities
- Syntax highlighting
- Programming language documentation
- Code debugging
- Source control management tools, such as Git, SVN, or Mercurial

Many modern IDEs support multiple programming languages. Using these IDEs, developers create applications in a variety of different languages, without needing to learn language-specific tooling, such as compilers and interpreters.

Developing Software with VS Code

VS Code is a popular open source IDE that supports multiple languages, including JavaScript, Python, Go, C#, TypeScript, and more. It provides syntax highlighting, code completion, debugging, and code snippet capabilities, along with a plug-in framework that allows you to install additional functionality from a plug-in marketplace.

In this course, you use VS Code to create, edit, and manage source code projects.

Overview of the VS Code Interface

The VS Code interface contains five primary components:

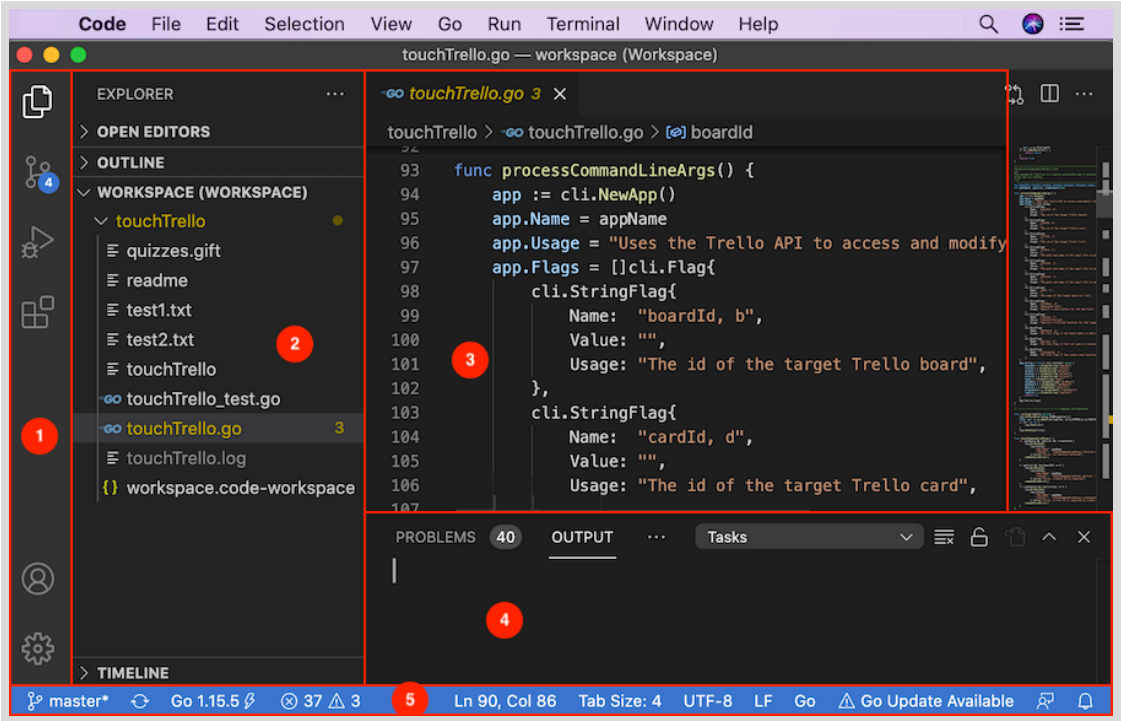


Figure 1.1: The VS Code user interface.

1. **The Activity Bar**. Located at the far-left, it contains shortcuts to change the view of the Side Bar. By default, the Activity Bar contains shortcuts for the Explorer, Search, Source Control, Debug, and Extensions views. You can also access these activity views from the `View` menu.

2. **Side Bar**. Located immediately to the right of the **Activity Bar**, this area displays the selected Activity view, such as the Explorer view.

3. **Editor Groups**. The top-right region of VS Code contains one or more groups of editors.

   By default, there is only one editor group. Any active editor takes up the entire region of the editor group.

   Click the `Split Editor Right` icon in the upper-right to create a second editor group. With two editor groups, you can edit two different files side by side.

4. **Panels**. Located below the editors, individual panels provide different output or debugging information for activities in VS Code. By default, four panels are provided:

   - Problems
   - Output
   - Debug Console
   - Terminal

5. **Status Bar**. Located at the bottom, this area provides information about the open project and files as you edit.

VS Code Workspaces

VS Code organizes a collection of related software projects and configuration settings in a **workspace**. Configuration data for each workspace is stored in a file with a `.code-workspace` extension.

From the `File` menu, you can close, save, and open a workspace.

For example, consider a web application, which is named `myapp`, with the following components:

- JavaScript code for the front end user interface of the application.
- Python code for the back-end application logic.
- Configuration and scripts for the application database.
- AsciiDoc files for the application documentation.

If you maintain each of these components in separate code repositories or folders, then you can add each folder to a `myapp` VS Code workspace that you dedicate for the application:

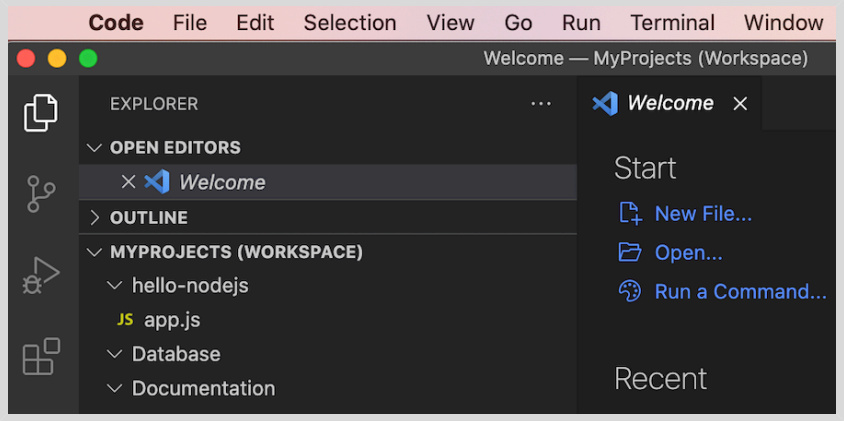


Figure 1.2: A VS Code workspace with multiple source code folders.

To add a source code folder to a workspace, click **File** > **Add Folder to Workspace...**.

To remove a source code folder from a workspace, access the Explorer view (**View** > **Explorer**). Right-click a selected top level workspace folder, and then select `Remove Folder from Workspace` to remove the folder from your workspace.

VS Code Integrated Terminal

Although VS Code integrates many development tools, you might need access to external development tools or applications. VS Code integrates the terminal from your operating system, enabling you to execute arbitrary terminal commands in VS Code. To view the integrated terminal, click **View** > **Terminal**.

You can open multiple terminals in the VS Code terminal panel. The terminal panel contains a list of open terminals. To display a terminal, select it from the list. To close a terminal, click **Kill Terminal**.

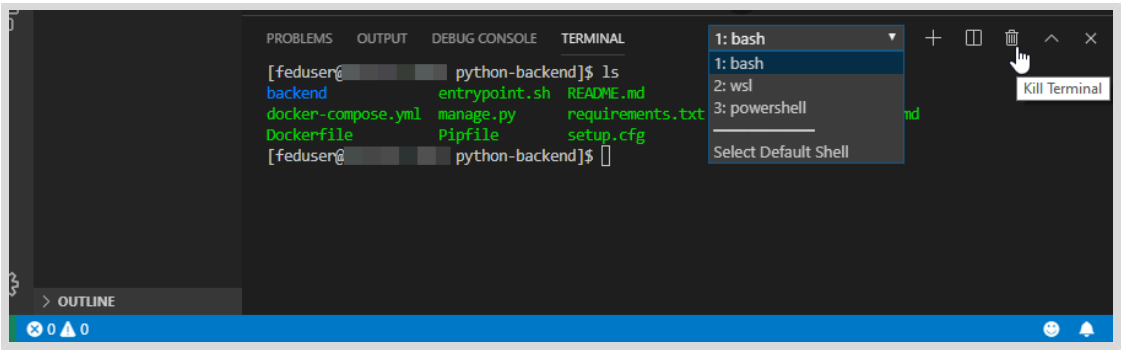


Figure 1.3: The VS Code integrated terminal

VS Code Extensions

Although the VS Code integrated terminal aids arbitrary command execution, you must install and learn to use any needed external commands. Additionally, terminal commands do not take advantage of common usage patterns in VS Code.

VS Code provides an extension framework to encourage the integration of software development capabilities. Any user or organization can contribute extensions. After an extension is developed, it is advertised and published on the VS Code marketplace.

You can search, download, and install extensions from the marketplace within VS Code. Click **View** > **Extensions** to access the Extensions view in the Side Bar.

Note
In this course, you do not install any additional extensions for VS Code.

Developing a Node.js Application Using VS Code

Many of the example web applications in the course exercises are Node.js applications. If an exercise in this course requires you to edit code, then use VS Code to make the changes.

Node.js is an open source runtime engine that executes JavaScript outside of a browser. It is designed to efficiently handle many concurrent connections for network applications. Additionally, Node.js enables you to write both front end and back end code in one language, JavaScript. For these reasons, Node.js is a popular runtime engine for web application development.

Installing Node.js

To install Node.js, navigate to https://nodejs.org/en/download in a browser. Click the appropriate link for your operating system, and then follow the instructions.

After you install Node.js, you use the `node` command to execute Node.js applications.

Node.js Modules

All modern programming languages support code reuse through shared libraries, packages, and modules.

In Node.js, **modules** are the smallest unit of reusable code. Node.js provides several built-in modules.

When you create a complex Node.js application, you write custom modules to group related logical code. Your custom modules are defined in JavaScript text files.

Use the built in `require` function to load a module. Consider the following example:

```
const http = require('http');
const mymodule = require('./mymodule');
```

The `http` variable is a reference to the built-in `http` module, while the `mymodule` variable is a reference to the module defined in the `mymodule.js` file.

Node.js Packages

Like other programming languages, Node.js provides a way to define and manage application dependencies. A Node.js application dependency is called a **package**. A package is a collection of one or more Node.js modules. Packages can be downloaded from a Node.js package repository.

Application dependencies are defined in the `package.json` file, located at the root of the Node.js project folder. An example `package.json` file follows:

```
{
    "name": "hello",
    "version": "0.1.0",
    "description": "An example package.json",
    "author": "Student <student@example.com>",
    "scripts": {
        "start": "node app.js"
    },
    "dependencies": {
        "cookie-parser": "1.4.*",
        "http-errors": ">=1.6.3",
    },
    "license": "MIT"
}
```

In this example, the `hello` application requires specific versions of the `cookie-parser` and `http-errors` packages.

The `package.json` file also defines other metadata for a Node.js application, such as the name, version, and author of the application.

The Node Package Manager

The Node Package Manager (NPM) is a command line tool used to create, install, and publish Node.js packages. For most operating systems, the `npm` command is included as part of the Node.js installation process.

To install the dependencies for a Node.js application, use the `npm install` command.

To initialize an empty directory as a Node.js project, use the `npm init` command. This creates a `package.json` file for a new Node.js application.

You can define additional scripts within the `scripts` section of `package.json`. For example, it is customary to define a script named `start` that starts the application. To run a script, pass

the name of the script to the `npm run` sub-command. For example, if you define a script named `build`, run that script with the following:

```
[user@host myapp]$ npm run build
```

Note

Some script names do not require the `run` sub-command. For example, `npm start` is the same as `npm run start`.

The Express Web Application Framework

Express is a common Node.js framework that aims to simplify the creation of web services. Because Express is a Node.js package, use the `npm install` command to install Express and save it as a dependency:

```
[user@host ~]$ npm install -S express
```

After installing the `express` package, Express is available to your application.

For example, the `express` package is a dependency for an application named `myapp`. The application has the following `package.json`:

```
{
  "name": "myapp",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "~4.16.1"
  }
}
```

Start the `myapp` application with the `npm start` command, which runs `node app.js`. The `app.js` file contains the primary application logic and contains the following:

```
const express = require('express');
const app = express();

app.get('/', function (req, res) {
  res.send('Hello, World!\n');
});

app.listen(8080, function () {
  console.log('Example app listening on port 8080!');
});

module.exports = app;
```

The `app` variable references an instance of an Express application. The application is configured to listen for requests on port 8080. When you access the application endpoint, the applications sends a response of `Hello, World!`.

References

Integrated Development Environment - Wikipedia
https://en.wikipedia.org/wiki/Integrated_development_environment

For more information about Visual Studio Code workspaces and project folders, refer to the Visual Studio Code documentation at
https://code.visualstudio.com/docs/editor/multi-root-workspaces

For more information about Visual Studio Code integrated terminal, refer to the Visual Studio Code documentation at
https://code.visualstudio.com/docs/editor/integrated-terminal

For more information about Node.js modules, refer to the Node.js documentation at
https://nodejs.org/api/modules.html#modules_modules

Guided Exercise

Developing Applications with VSCode

In this exercise, you will use Visual Studio Code (VS Code) to create a simple Node.js application.

Outcomes

You should be able to:

- Download and install Node.js.
- Download and install VS Code.
- Create a workspace in VS Code.
- Add a project folder to a VS Code workspace.

Before You Begin

To perform this exercise, ensure that:

- You have access to a Linux (Debian or Fedora-based), macOS, or Windows system, including the required permissions to install software on that system.

Instructions

1. Download and install the long term support (LTS) version of `Node.js`.

- On Windows, download and install the latest LTS version [https://nodejs.org/dist/v16.13.0/node-v16.13.0-x64.msi].
- On macOS, downlad and install the latest LTS version [https://nodejs.org/dist/v16.13.0/node-v16.13.0.pkg].
- On Linux, use the documentation [https://nodejs.org/en/download/package-manager/] to download and install the latest package that contains Node.js.

2. Download and install `VS Code`.

Use the appropriate installation for your device from the VS Code Download site [https://code.visualstudio.com/download].

3. Open VS Code and create a workspace to host your projects.

3.1. Open the VS Code application according to your operating system. Click **View** > **Explorer** to display the Explorer view.

3.2. If you installed and used VS Code on your system previous to this course, then click **File** > **Close Workspace**. If **File** > **Close Workspace** is not available, then skip this step.

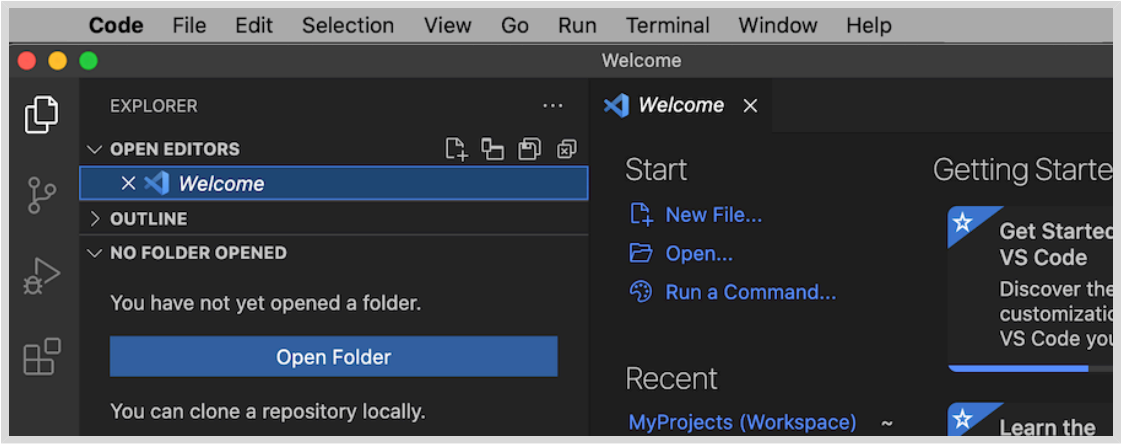


Figure 1.4: The VS Code application

3.3. Click **File** > **Save Workspace As ...**. In the window that displays, navigate to your home directory. Type `My Projects` as the file name and then click **Save**. The Explorer view displays a **Add Folder** button to add project folders to your workspace.

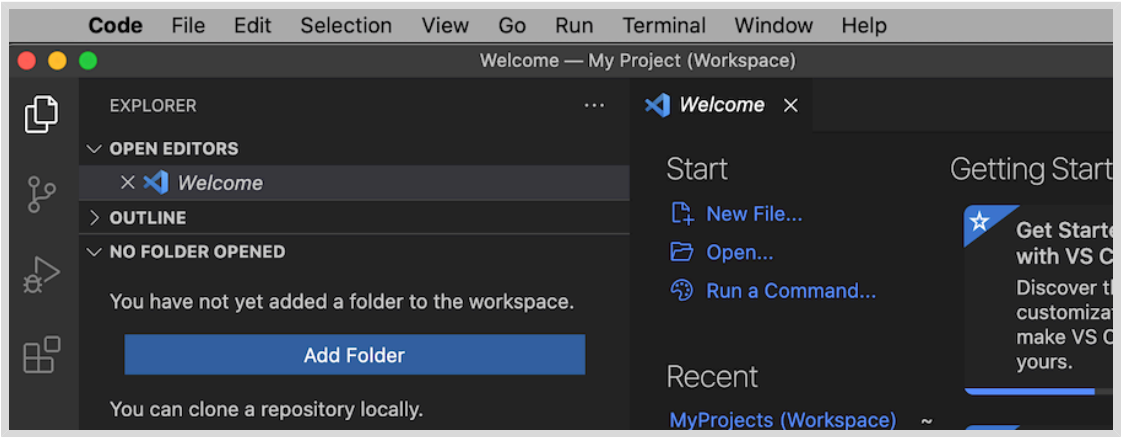


Figure 1.5: The VS Code workspace.

4. Create a `hello-nodejs` project folder, and then add it to your workspace.

4.1. In VS Code, click **File** > **Add Folder to Workspace...**. In the window that displays, navigate to your home directory. Create a new folder named `hello-nodejs`. Click **Add** to add this new folder to your workspace.

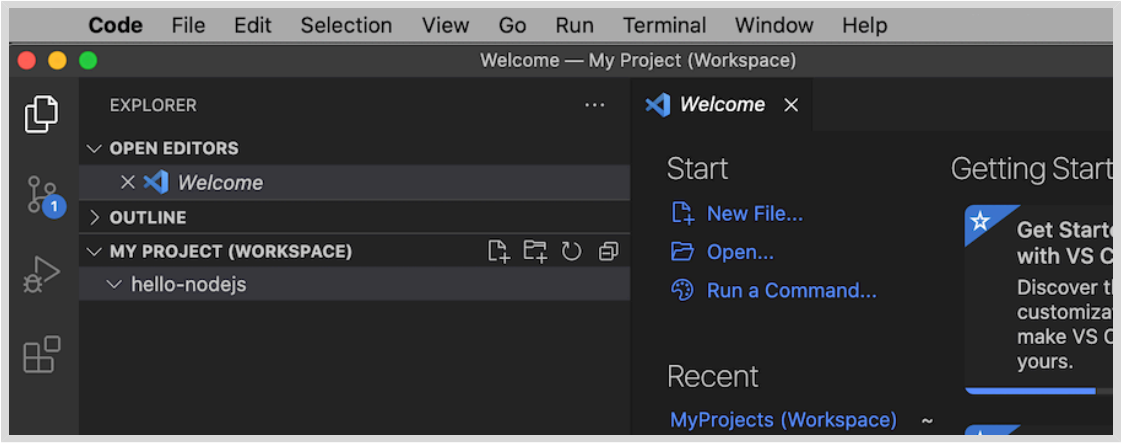


Figure 1.6: Add a folder to the VS Code workspace.

5. Create a `app.js` file in the project. Save the file with the following content:

```
console.log("Hello World!\n");
```

5.1. Right-click `hello-nodejs` in the workspace, and then select `New File`. Enter `app.js` for the file name to launch a VS Code tab for the new file.

5.2. Add the text `console.log("Hello World!\n");` to the `app.js` editor tab, and then save the file (**File** > **Save**).

6. Right-click `hello-nodejs` in the workspace, and then select `Open in Integrated Terminal` to access the `hello-nodejs` project from the VS Code integrated terminal.

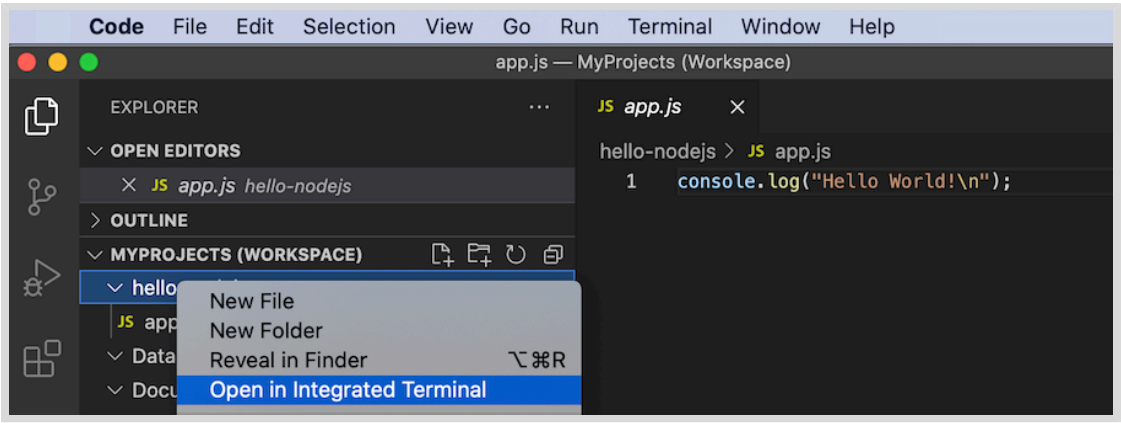


Figure 1.7: Open a project folder in the VS Code integrated terminal.

7. In the integrated terminal, execute `node app.js` to test your sample code and your Node.js installation.

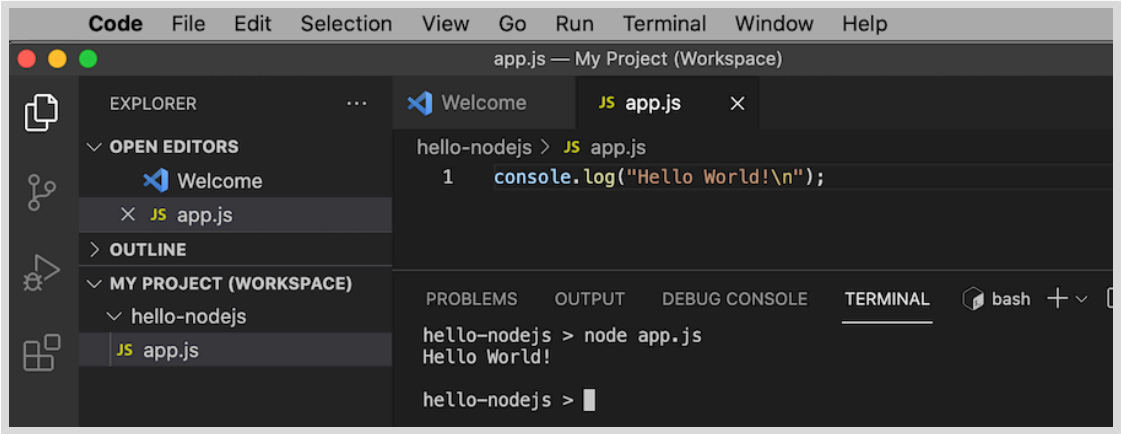


Figure 1.8: Execute a Node.js application in the VS Code intergrated terminal.

8. To clean up your work, click the **Kill Terminal** icon to close the integrated terminal window.

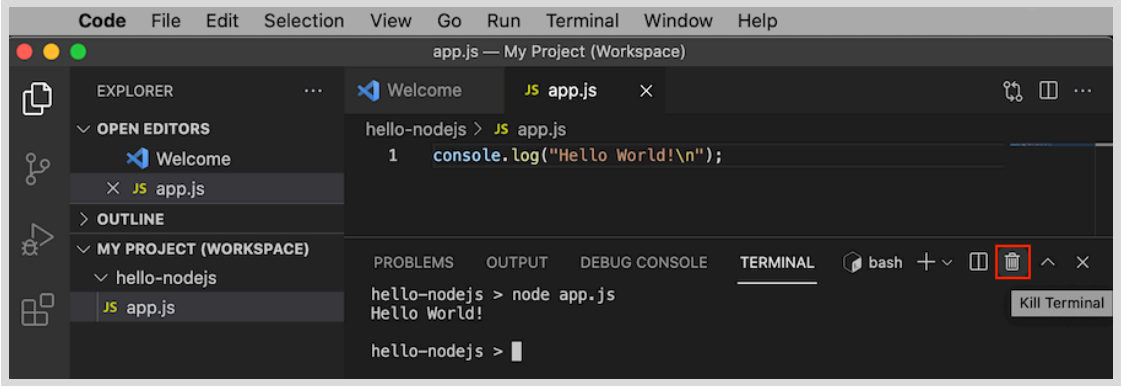


Figure 1.9: Closing the integrated terminal window in VS Code.

Finish

This concludes the guided exercise.

Initializing a Git Repository

Objectives

After completing this section, you should be able to create a Git repository.

Software Version Control

A Version Control System (VCS) enables you to efficiently manage and collaborate on code changes with others. Version control systems provide many benefits, including:

- The ability to review and restore old versions of files.
- The ability to compare two versions of the same file to identify changes.
- A record or log of who made changes at a particular time.
- Mechanisms for multiple users to collaboratively modify files, resolve conflicting changes, and merge the changes together.

There are several open source version control systems available including:

- CVS
- SVN
- Git
- Mercurial

Introducing Git

Git is one of the most popular version control systems. For this reason, you use Git as the version control system for all the exercises in this course.

Git can convert any local system folder into a Git repository. Although you have many of the benefits of version control, your Git repository only exists on your local system. To share your repository with another collaborator, you must host the repository on a code repository platform.

There are many code repository platforms, including:

- GitHub
- GitLab
- BitBucket
- SourceForge

In this course, you use GitHub to host and share your Git repositories.

Git Workflow Overview

To retrieve the project files for an existing software project with Git, you **clone** the Git repository for the project. When you clone a project, a complete copy of the original remote repository is created locally on your system.

Your local copy of the repository contains the entire history of the project files, not just the latest version of project files. You can switch to different versions of the files, or compare two different versions of a file, without connecting to the remote repository. This allows you to continue implementing code changes when the remote repository is not available.

Git does not automatically synchronize local repository changes to the remote repository, nor does it automatically download new remote changes to your local copy of the repository. You control when Git downloads changes from the remote repository, and when local changes are uploaded to the remote repository.

To track file changes in Git, you create a series of project snapshots. Each snapshot is called a **commit**. When you commit code changes to your repository, you create a new code snapshot in your Git repository.

Each commit contains metadata to help you find and load this snapshot at a later time:

- **Commit message** - A high level summary of the file changes in the commit.
- **Timestamp** - The date and time that the commit was created.
- **Author** - The creator or creators of the commit.
- **Commit hash** - A unique identifier for the commit. A commit hash consists of 40 hexadecimal characters. If a Git command requires a commit hash to perform an operation, then you can abbreviate the commit to enough characters to be unique within the repository.

After you create commits in a local repository on your system, you must **push** your changes to the remote repository. When you push changes to a remote Git repository, you upload local commits to the remote repository. After a push, your commits are available for others to download.

When other contributors push commits to a remote repository, those commits are not present in your local repository. To download new commits from other contributors, you **pull** changes from the remote Git repository.

Installing Git

Git is an open source version control system that is available for Linux, macOS, and Windows systems. Before you can use Git, you must install it.

In a browser, navigate to https://git-scm.com/downloads and follow the directions for your operating system. After installing Git on your system, you can use VS Code to manage your Git source code repositories.

To test your Git installation, open VS Code and access the integrated terminal (**View** > **Terminal**). At the terminal prompt, execute `git --version`. If Git is correctly installed, then a version number prints in the terminal:

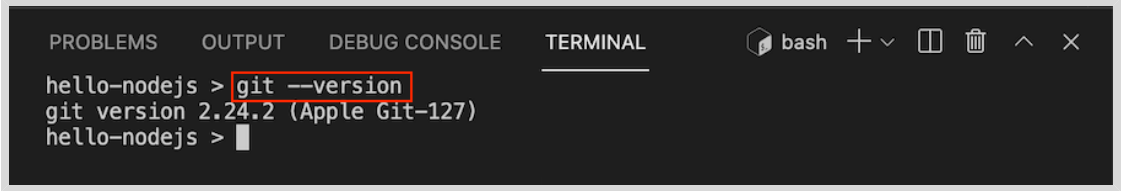


Figure 1.10: Testing the installation of Git in VS Code

The Source Control View in VS Code

Use the VS Code Command Palette (**View** > **Command Palette...**) and the Source Control view (**View** > **SCM**) to execute Git operations, such as cloning a repository or committing code changes.

By default, the VS Code Source Control view is different when you have one Git repository in your workspace, compared to when you have multiple Git repositories in your workspace. When you have multiple Git repositories in your workspace, then the Source Control view displays a `SOURCE CONTROL` list:

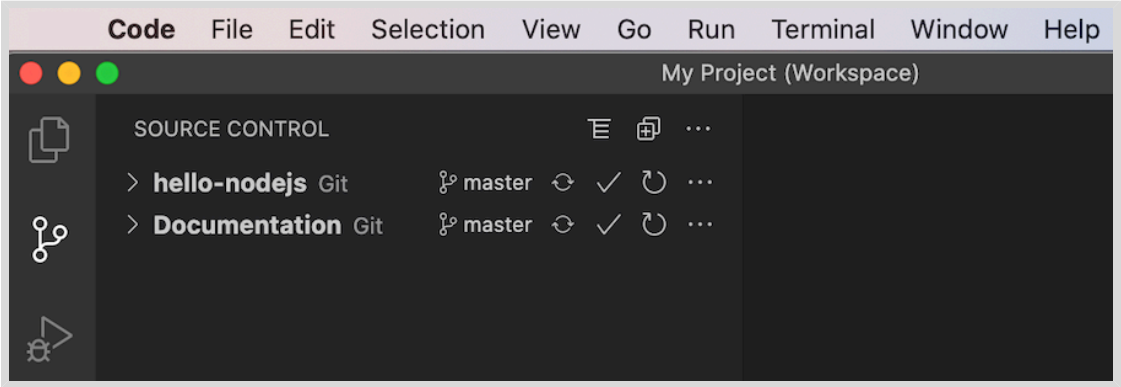


Figure 1.11: The source control list in the source control view of VS Code.

By default, when there is only a single Git repository in your workspace, then the Source Control view does not display the `SOURCE CONTROL` list.

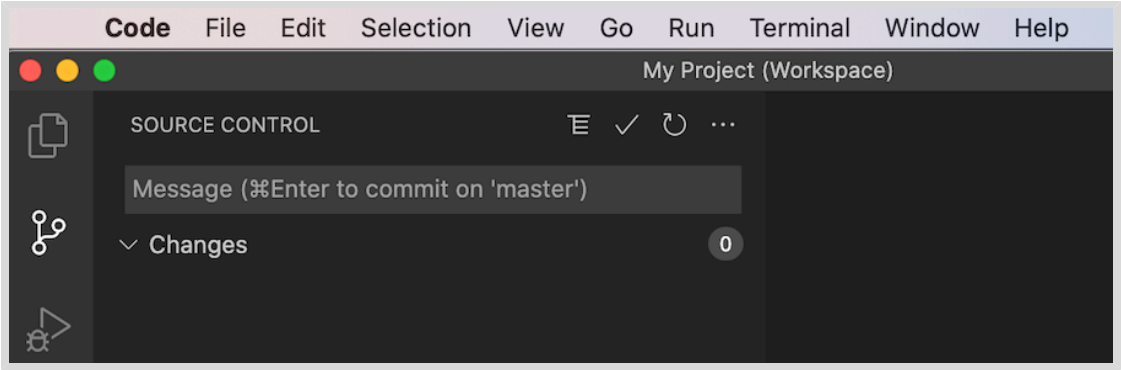


Figure 1.12: Source control view for a single Git repository.

For a consistent user interface, independent of the number of Git repositories in your workspace, you must enable the `Always Show Repositories` source control management option. To enable this option, access the Command Palette (**View** > **Command Palette...**) and type `settings`. Select `Preferences: Open Settings (UI)` from the list of options. VS Code displays a settings window:

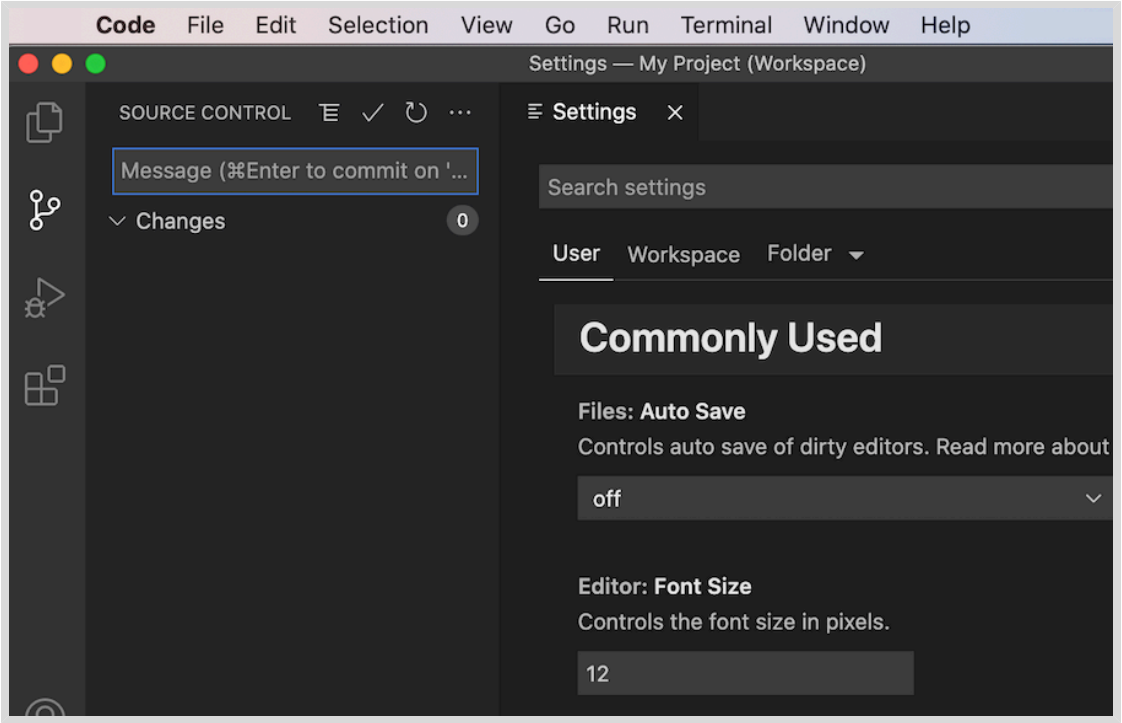


Figure 1.13: The settings window for VS Code.

Click **User**, and then click **Features** > **SCM**. VS Code displays Source Control Management (SCM) options for VS Code. Select `Always Show Repositories`.

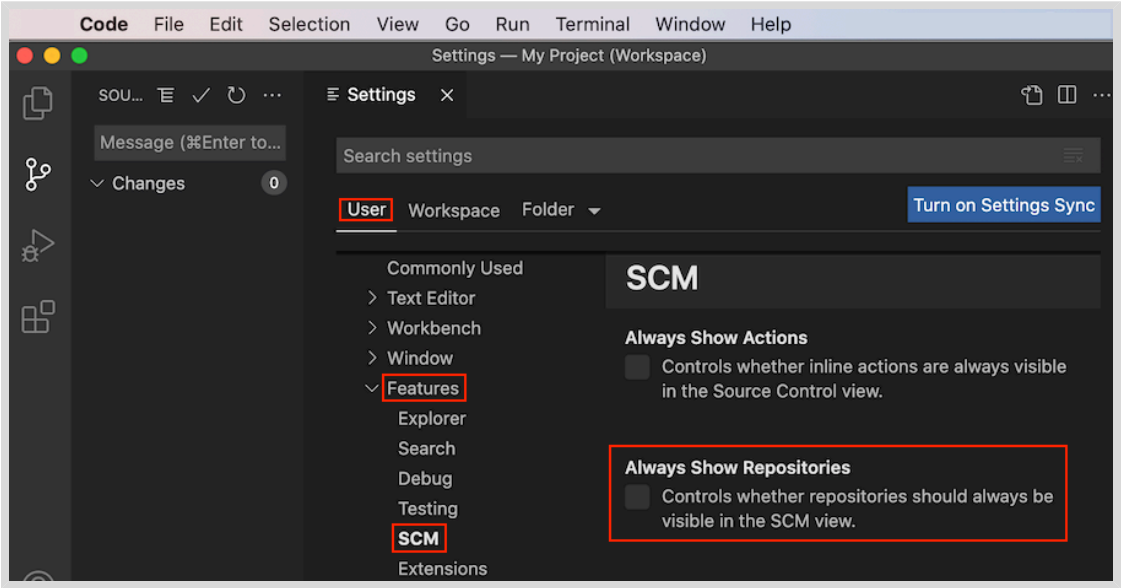


Figure 1.14: Option to always show the source control repositories list in the source control view of VS Code.

When you enable this option, then VS Code displays the `SOURCE CONTROL` list in the Source Control view for any number of workspace Git repositories, including only one repository.

Cloning a Git Repository

Use the VS Code Command Palette (**View** > **Command Palette...**) and the Source Control view (**View** > **SCM**) to execute Git operations, such as cloning a repository or committing code changes.

To clone a remote Git repository in VS Code, access the Command Palette (**View** > **Command Palette...**). Type `clone` at the prompt, and then select `Git: Clone` from the list of options.

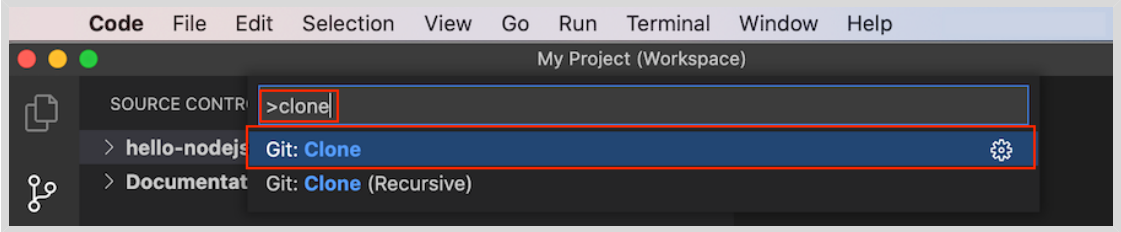


Figure 1.15: Using the command palette to clone a repository.

VS Code prompts you for the URL of the remote repository, and then prompts for the desired location of the local repository on your file system.

After VS Code clones the repository, add the cloned repository folder to your VS Code workspace.

Committing Code Changes

After adding a cloned repository to your VS Code workspace, you can edit project files like any other workspace files. As you edit project files, Git assigns a status to each file:

- **Modified** - The file differs from the most recent version. Modified files are not automatically added or committed to your Git repository.
- **Staged** - A staged file is a modified file that you flag to be included as part of your next code commit to the repository.

When you commit code to the repository, only those files with a **staged** status are included in the commit. If your project contains modified files that are not staged, then those files are not included in the commit. This feature enables you to control the file changes that are included in each commit.

In VS Code, the Source Control view (**View** > **SCM**) displays all modified and staged repository files. After you save edits to a file, the file name displays in the `CHANGES` list.

To add a modified file to your next code commit, click the modified file in the `CHANGES` list, from the Source Control view. VS Code displays a new tab to highlight the changes to the file:

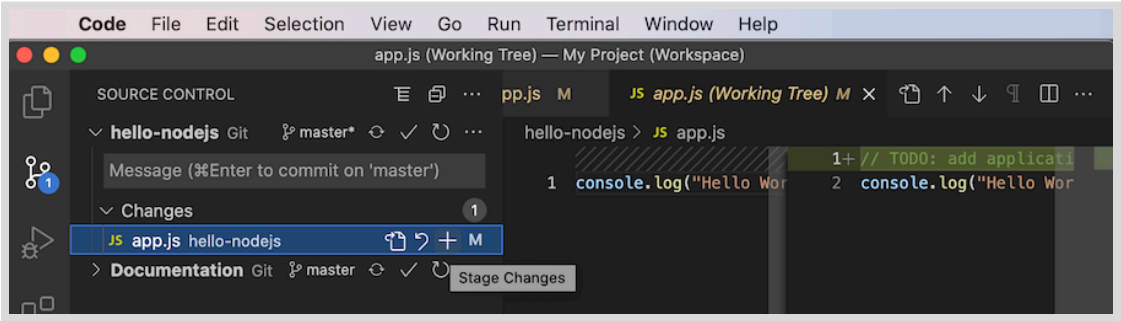


Figure 1.16: Review file changes before staging a file.

If the file changes are accurate and complete, click the plus button next to the file to stage changes, which adds the file changes to your next code commit.

After all of your desired file changes are staged, provide a commit message in the Source Control view. Then, select the check box to commit all of the staged file changes to your local repository:

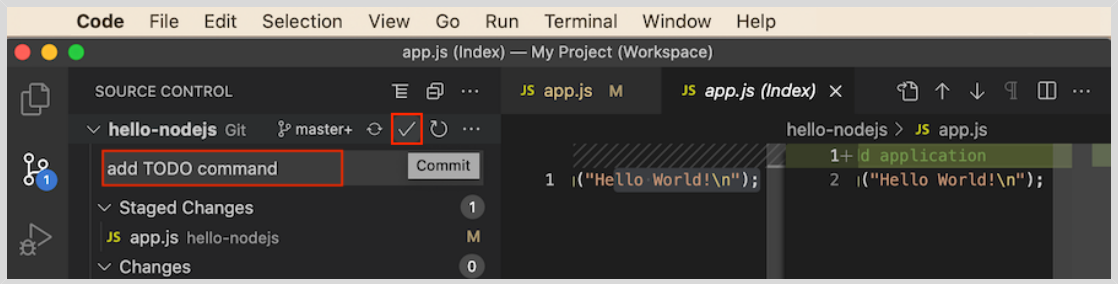


Figure 1.17: Commit staged files in VS Code.

Using a Remote Repository

When you commit code changes, you only commit code to your local repository. No changes are made to the remote repository.

When you are ready to share your work, synchronize your local repository to the remote repository. To publish local commits to the remote repository, Git performs a push operation. To retrieve commits from the remote repository, Git performs a pull operation. VS Code handles the pull and push Git operations when you synchronize your local repository to the remote repository.

The Source Control view compares your local repository with the corresponding remote repository. If there are commits to download from the remote repository, then the number of commits displays with a download arrow icon. If there are local commits that you have not published to the remote repository, then the number of commits appears next to an upload arrow icon. This is the synchronize changes icon, highlighted in the following figure.

The icon shows there are zero commits to download and one commit to upload to the remote repository:

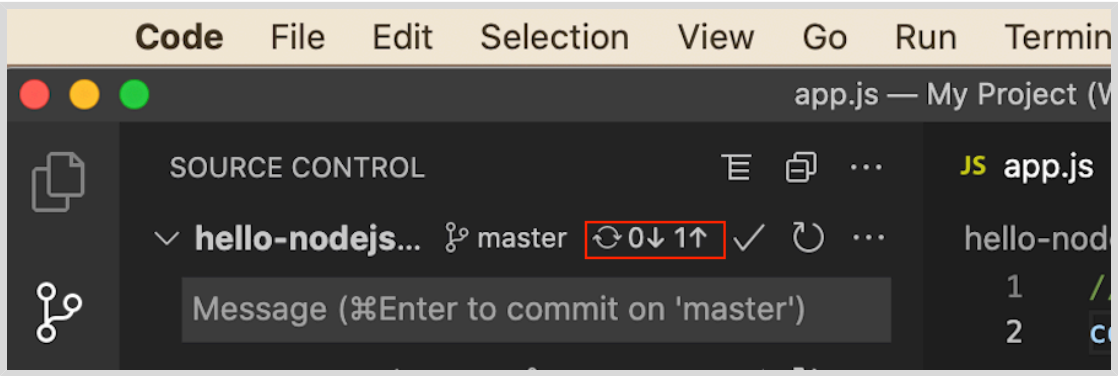


Figure 1.18: Git repository status in the source control view and status bar.

Note

Because Git does not run in the background, VS Code and Git will not be aware of available commits to download until *some* synchronize action is performed. In general, do not assume that these number are always accurate.

Click the synchronize changes icon to publish your local code commits to the remote repository. Alternatively, you can click the same icon in the status bar to publish your code commits.

Important

If the Source Control view (**View** > **SCM**) does not contain a `SOURCE CONTROL REPOSITORIES` heading, then you may not see a the synchronize changes icon.

To enable it, right-click `SOURCE CONTROL`. If a check mark does not display to the left of `Source Control Repositories`, then click `Source Control Repositories`.

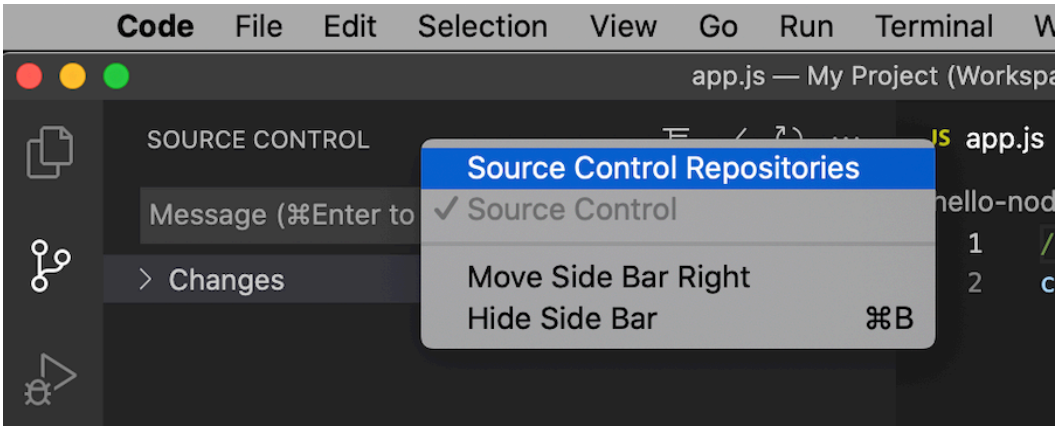


Figure 1.19: Enable the source control repositories listing.

Adding a Github Access Token

If you attempt to push a commit to a remote repository for the first time, VS Code redirects you to a browser window to login into Github to provide access. After logging into Github, generate a personal access token for VSCode to use for your repository. A personal access token is required to be used in place of a password with the command line or with the Github API.

To create a personal access token take the following steps:

1. After logging into Github, select your profile icon in the upper right hand corner and click (**Your Profile Icon** > **Settings**).

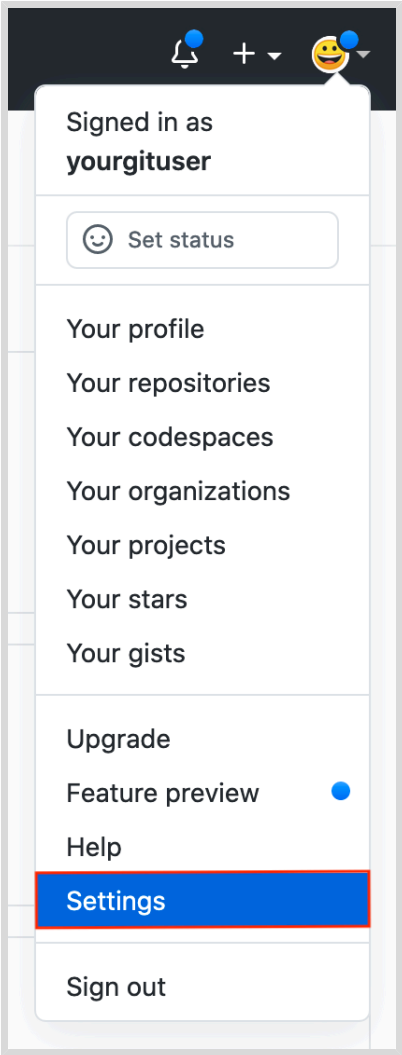


Figure 1.20: Github profile settings.

2. On the left sidebar, click `Developer settings`.

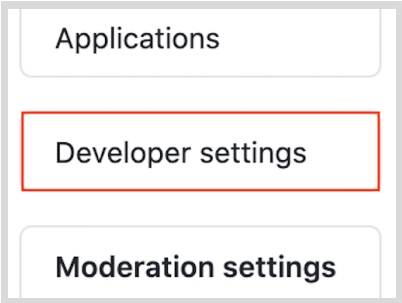


Figure 1.21: Github developer settings.

3. In the left sidebar, click `Personal access tokens`.

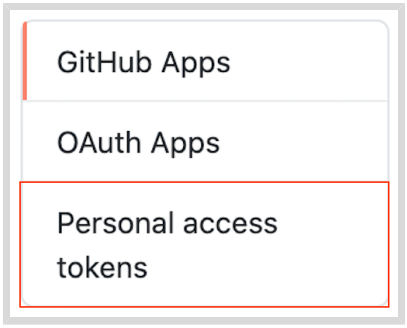


Figure 1.22: Github personal access token.

4. Click `Generate new token`.

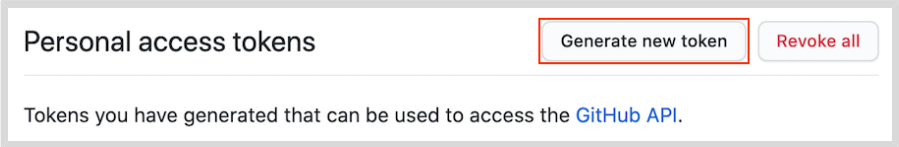


Figure 1.23: Generate new GitHub token.

5. In the `Note` field, give your token a descriptive name. Select an appropriate `Expiration` date for you token.

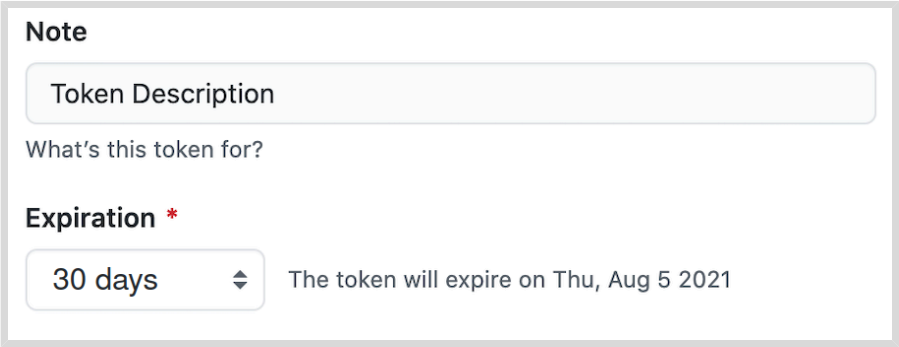


Figure 1.24: Token note and expiration

6. Select the scopes, or permissions, you would like to grant this token. To use your token to access repositories from the command line, select `repo`.

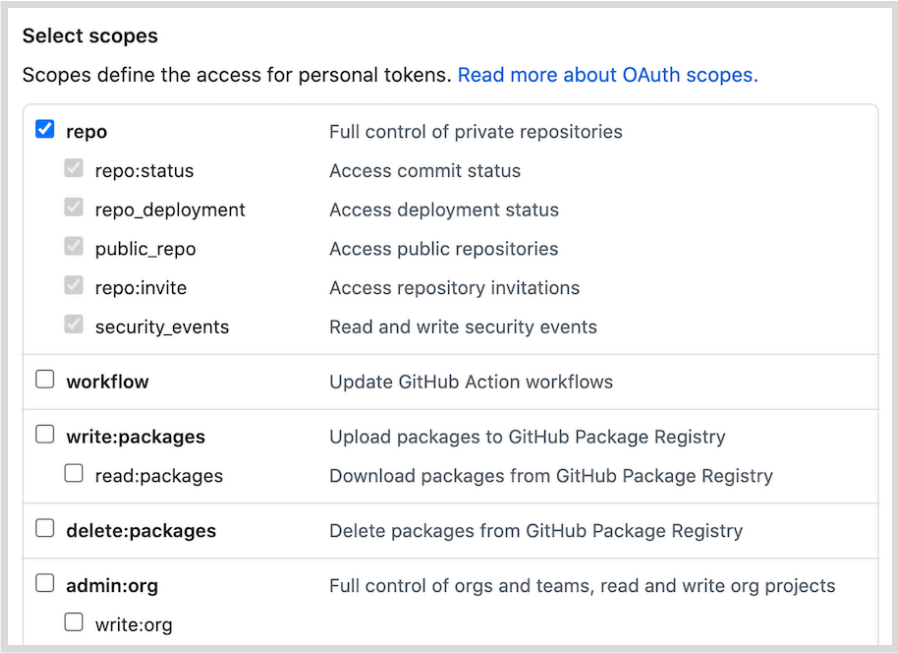


Figure 1.25: Token scope and permissions.

7. Towards the bottom of the page, click `Generate token`.

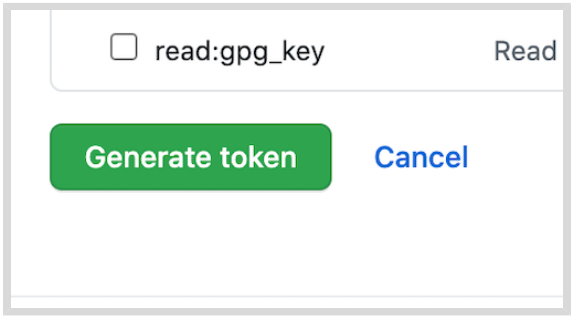


Figure 1.26: Generate the GitHub token.

8. Copy the generated token and provided it when prompted by VS Code. Alternatively, if you are asked for username and password, use the token in the username field and leave the password field empty.

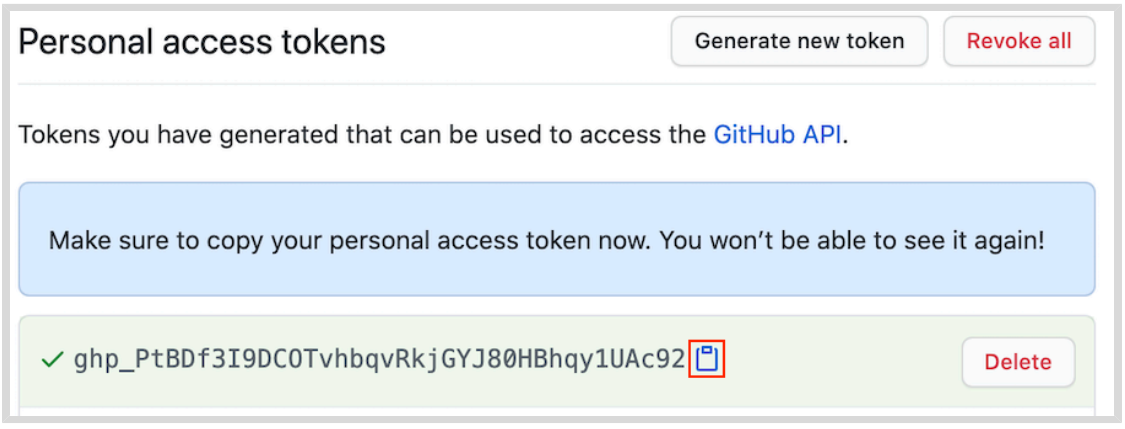


Figure 1.27: Copy the generated token.

9. Add the personal access token to the VS Code prompt.

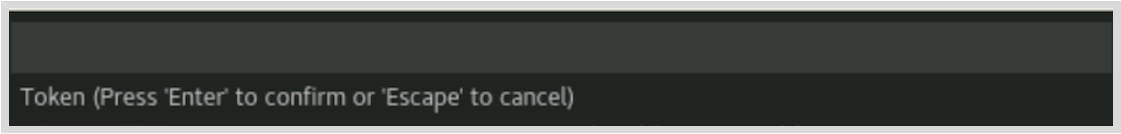


Figure 1.28: Use the Generated token.

Initializing a New Git Repository

If you have an existing software project that needs version control, then you can convert it to a Git repository.

To convert any file system folder into a Git repository, access the VS Code Command Palette (**View** > **Command Palette...**). Type `intialize` at the prompt, and then select `Git: Initialize Repository` from the list of options.

VS Code displays a list of project folders in the workspace.

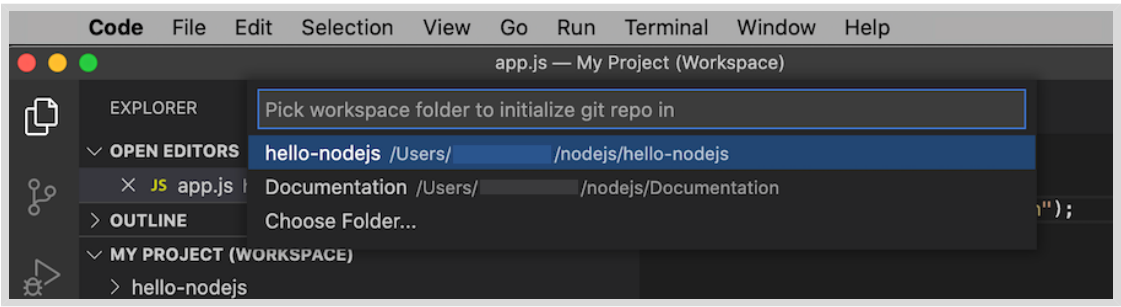


Figure 1.29: Using VS Code to initialize a Git repository.

After you select a project folder, Git initializes the folder as an empty Git repository. The Source Control view displays an entry for the new repository. Because the folder is initialized as an empty repository, every file in the project folder is marked as an untracked file.

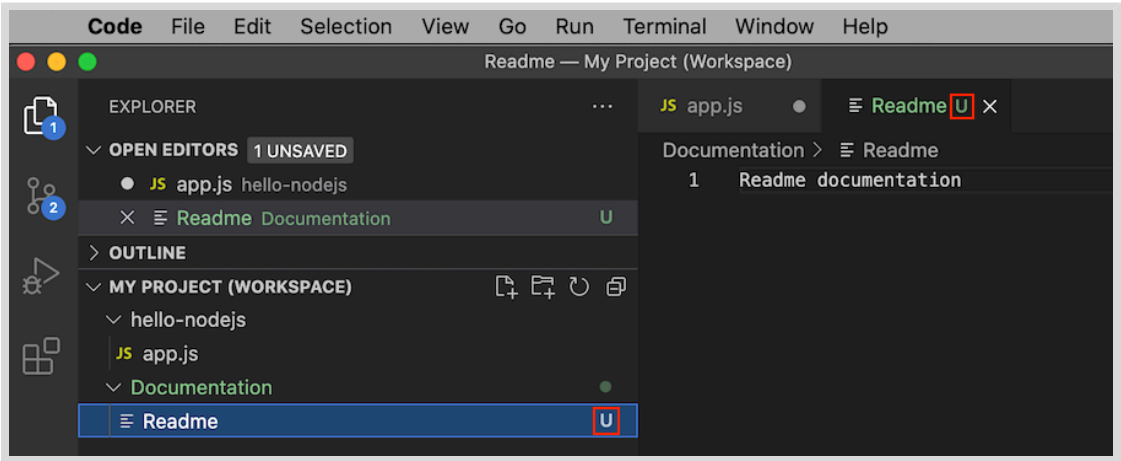


Figure 1.30: Untracked files in a Git repository.

For any project file that requires version control, click the plus icon to add the file to the local repository. Each added file displays in the `STAGED CHANGES` list in the Source Control view. After you stage all of the project files, provide a commit message, and then click the check mark icon to create the first commit in the repository.

When you create a new repository from a local file system folder, the new repository is not associated with a remote repository. If you need to share your repository:

1. Create a new Git repository on a code hosting platform, such as GitHub.

2. Associate your local repository to the new remote repository, and then synchronize changes.

Adding a Remote Repository to a Local Repository

After you create a new repository on a code hosting platform, the platform provides you with a HTTPS and SSH URL to access the repository. Use this URL to add this hosted repository as a remote repository for your local repository.

Note

In this course, you only use HTTPS URLs to access remote code repositories. HTTPS access to a Git repository requires very little additional configuration, but does require that you provide credentials for the code hosting platform.

You can configure your Git installation to cache your credentials. This helps minimize re-entering credentials each time you connect to the remote repository.

SSH access to Git repository requires the configuration of your SSH keys with the code hosting platform.

SSH key configuration is beyond the scope of this course.

Access the VS Code Command Palette to add a remote repository to your local repository. Type `add remote` at the prompt, and then select `Git: Add Remote` from the list of options. If you are prompted to select a local repository, then make an appropriate selection.

At the prompt, enter `origin` for the remote name; `origin` is the conventional name given to the remote repository that is designated as the central code repository.

At the prompt, enter the HTTPS URL for your remote repository. If you have commits in your local repository, a `Publish Changes` icon displays in the `SOURCE CONTROL` list.

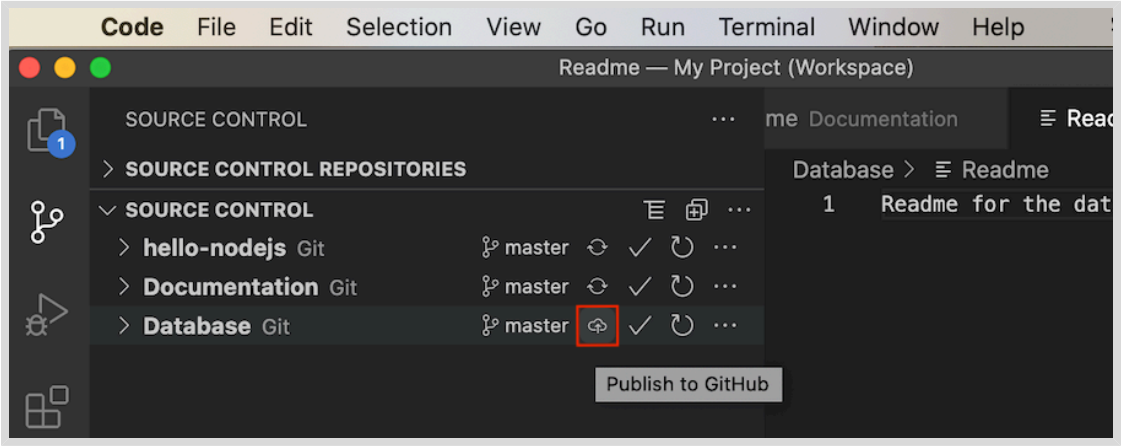


Figure 1.31: Publish a local Git repository to a remote repository.

Click the `Publish to GitHub` icon to push your local commits to the remote repository. If you are prompted, then provide the necessary remote repository credentials.

References

For more information about installing Git, refer to the Git documentation at
https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Git Basics
https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository

For more information about using Git and version control in VS Code, refer to the VS Code documentation at
https://code.visualstudio.com/docs/editor/versioncontrol

For more information about using a personal access token, refer to the Github documentation at
Creating a personal access token
https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/creating-a-personal-access-token

Guided Exercise

Initializing a Git Repository

In this exercise, you will use VS Code to push your project source code to a remote Git repository.

Outcomes

You should be able to:

- Install Git.
- Initialize a local folder as a Git repository.
- Stage a file in a Git repository.
- Commit staged files to a local Git repository.
- Push commits in a local Git repository to a remote repository.

Before You Begin

To perform this exercise, ensure that:

- You have access to a Linux (Debian or Fedora-based), macOS, or Windows system and the required permissions to install software on that system.
- Visual Studio Code (VS Code) is installed on your system.

Instructions

1. Download and install `Git`.

    1.1. Linux Installation.

    - Open a new command line terminal.
    - To install `Git` on Ubuntu and Debian systems, use the following command.

```
[yourname@yourhost ~]$  sudo apt install git
```

The command may prompt for your password to install the package.

- To install `Git` on Fedora and Red Hat Enterprise Linux 8 systems, use the following command:

```
[yourname@yourhost ~]$ sudo dnf install git
```

The command may prompt for your password to install the package.

1.2. macOS Installation.

- Git is installed by default on the latest macOS versions. To verify the Git installation, open a new command line terminal and enter the following command:

```
$ git --version
git version 2.24.2 (Apple Git-127)
```

Note

The Git version on your system might differ. Any recent version of Git will work with this course.

1.3. Windows Installation.

- In a browser on your Windows system, navigate to https://git-scm.com/download/win and save the executable file to your system.
- In Windows Explorer, navigate to the downloaded file. Double-click the file to start the setup wizard. If prompted, click **Yes** to allow the installer to make changes to your system.
- Click **Next** to accept the license agreement.
- Click **Next** to accept the default installation location for Git. If a window displays a warning about the installation location, click **Yes** to continue the installation of Git to that location.
- Click **Next** to accept the installation of the default set of components.
- Click **Next** to accept the default Start Menu Folder.
- Select `Use Visual Studio Code as Git's default editor` from the editor list to use VS Code as the default editor. Click **Next**.
- At the `Adjusting your PATH environment` prompt, click **Next**.
- Make an appropriate choice for the HTTPS transport back-end. If you are unsure of which option to select, then accept the default selection. Click **Next**.
- At the `Configuring the line-ending conversions` prompt, accept the default selection and click **Next**.
- Click **Next** to accept the default terminal emulator settings.
- At the `Configuring extra options` prompt, click **Next** to accept the defaults.
- Click **Install** to accept the default experimental features and begin installation. Wait for installation to complete, and then proceed to the next step.
- Click **Finish** to exit the setup wizard.

2. Use VS Code to test your Git installation. Configure your Git installation identity with your GitHub credentials.

2.1. Open VS Code.

2.2. Click **Terminal** > **New Terminal** to open an integrated terminal.

2.3. Execute `git --version` in the integrated terminal to test the installation of Git. The command prints the version of the Git installation on your system.

> **Note**
> VS Code depends on the configuration options selected during the Git installation process. If the `git --version` command fails in the integrated terminal, try restarting VS Code. Then, repeat this step to check the installation of Git.

2.4. In a browser, navigate to https://github.com. If you do not have a GitHub account, then create one. Log in to GitHub.

2.5. In the VS Code integrated terminal, execute `git config --global user.name` *`yourgituser`*, replacing *yourgituser* with your GitHub user name.

2.6. In the VS Code integrated terminal, execute `git config --global user.email` *`user@example.com`*, replacing *user@example.com* with the email address associated with your GitHub account.

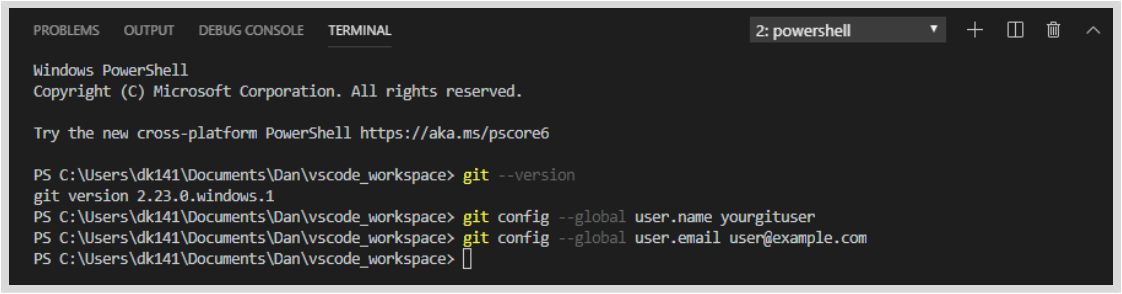


Figure 1.32: The VS Code integrated terminal.

> **Note**
> Git requires your GitHub user name and password for certain transactions with remote repositories.
>
> On Windows systems, Git manages these credentials by default. You are only prompted for credentials the first time you connect to a remote repository.
>
> By default on Linux and macOS systems, Git does not manage your remote repository credentials. Git prompts for your credentials each time you connect to GitHub.
>
> To cache your credentials on Linux or macOS systems, execute the following command from a system terminal:

```
$> git config --global credential.helper cache
```

3. Enable the `Always Show Repositories` source control management option in VS Code.

3.1. Access the Command Palette (**View** > **Command Palette...**) and type `settings`. Select `Preferences: Open Settings (UI)` from the list of options.

3.2. When the `Settings` window displays, click **User** > **Features** > **SCM**.

3.3. VS Code displays Source Control Management (SCM) options for VS Code. Select `Always Show Repositories`.

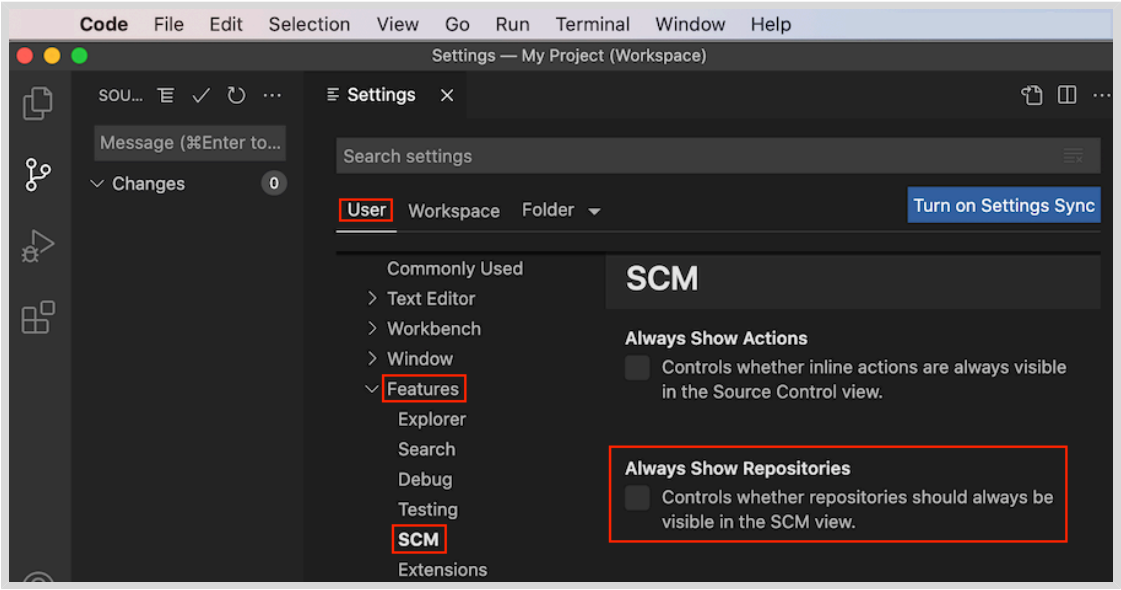


Figure 1.33: Option to always show the source control repositories list in the source control view of VS Code.

3.4. Close the `Settings` tab.

4. Ensure that you have a `hello-nodejs` project folder in your VS Code workspace. If you already have a `hello-nodejs` project folder in your VS Code workspace from a previous exercise, then skip this step.

4.1. Download the following zip file to your system:

https://github.com/RedHatTraining/DO101x-apps/releases/download/v0.1/hello-nodejs.zip.

Unzip the file, which creates a `hello-nodejs` folder on your system. The `hello-nodejs` folder contains a single file, `app.js`. Note the location of the `hello-nodejs` folder. You use this folder in a later step.

4.2. Click **File** > **Add Folder to Workspace...**

4.3. In the file window, navigate to the location of the unzipped `hello-nodejs` folder. Select the `hello-nodejs` folder and click **Add**.

5. Initialize the `hello-nodejs` project as a Git repository.

5.1. Access the VS Code Command Palette (**View** > **Command Palette...**).

5.2. Type `initialize`. VS Code provides a list of possible commands that match what you type. Select `Git: Initialize Repository` from the list of Command Palette options.

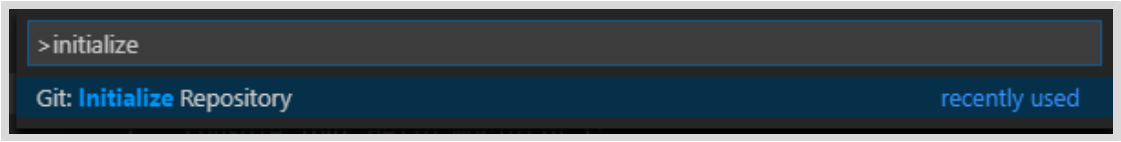


Figure 1.34: Git repository initialization using the command palette.

5.3. Select `hello-nodejs` from the list of workspace folders.

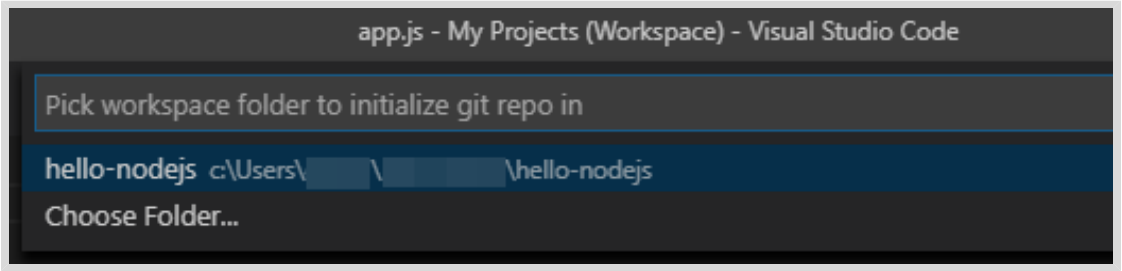


Figure 1.35: Selection prompt to initialize a local Git repository.

6. Create a commit from the `app.js` file.

6.1. Click **View** > **SCM** to access the `Source Control` view in the Activity Bar.

6.2. Hover over the `app.js` entry under `CHANGES`. VS Code displays a message that the `app.js` file is untracked. Click the plus sign for the `app.js` entry to add the file to the repository.

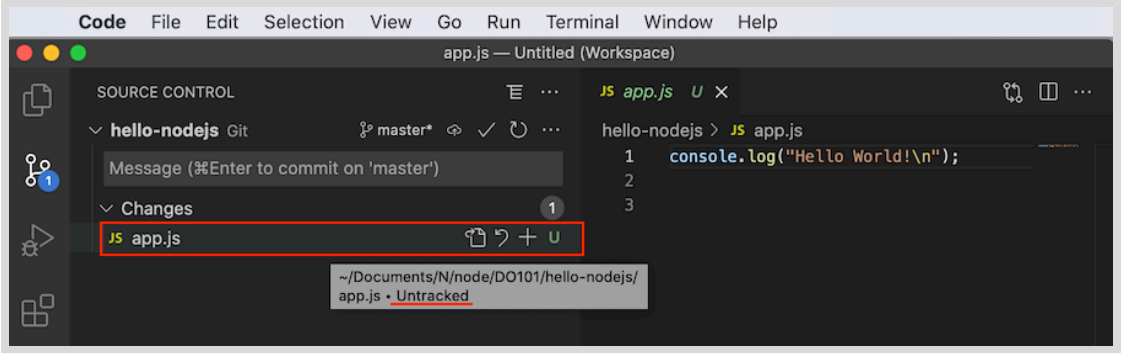


Figure 1.36: List of changed files.

This stages the `app.js` file for the next commit.

The file now appears under the `STAGED CHANGES` heading.

6.3. Click in the `Message (press Ctrl+Enter to commit)` field. Type `add initial app.js code` in the message field. Click the check mark icon to commit the changes.

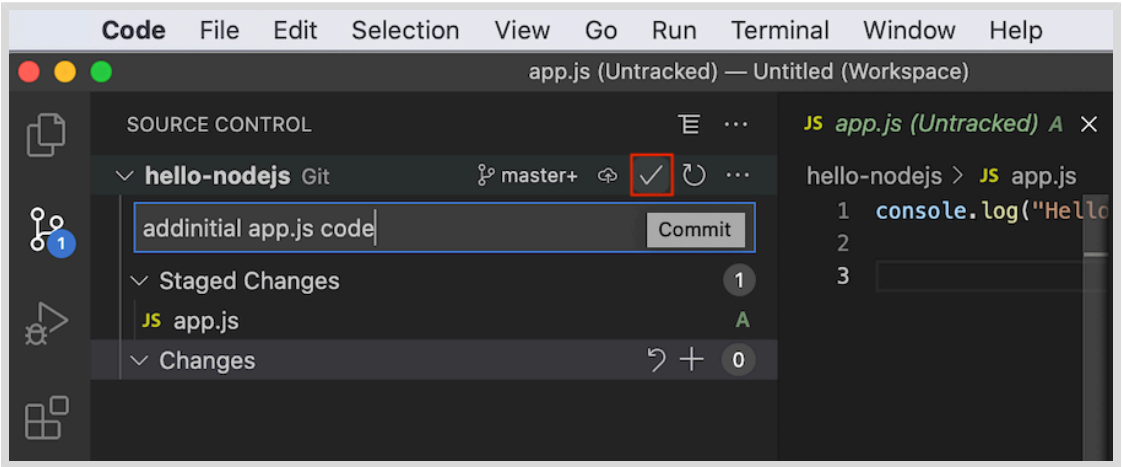


Figure 1.37: VS Code commit message box.

7. Create a new GitHub repository to host your project files. Add an access token for access to your git repository.

7.1. In a browser, navigate to https://github.com. If you are not logged in to Github, then log in.

7.2. Click the + on the upper-right, and then select `New repository` from the list displayed.

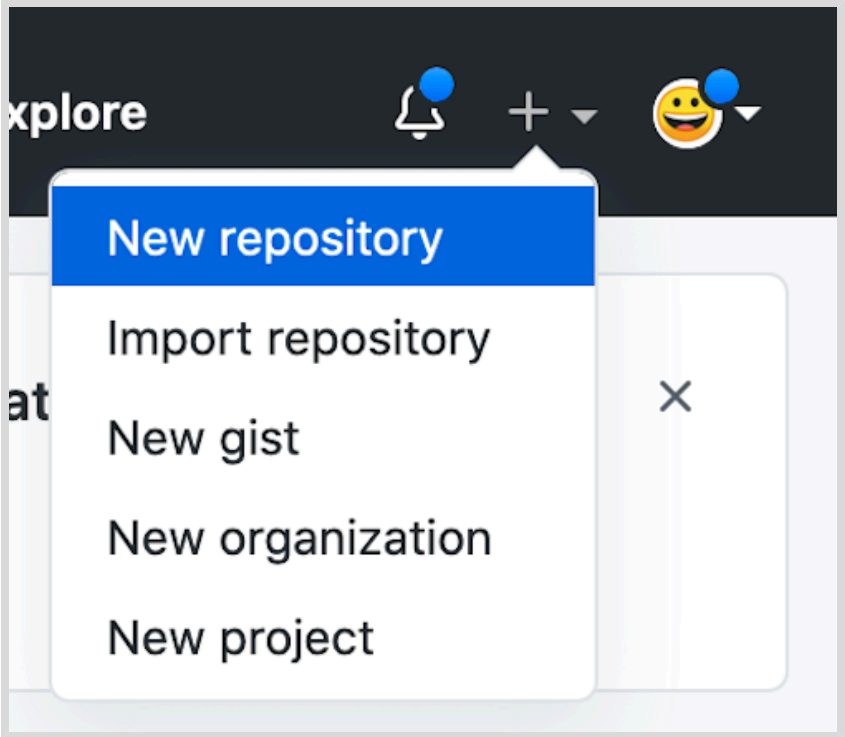


Figure 1.38: Create a new Git repository on GitHub.

7.3. Type `hello-nodejs` in the `Repository name` field. By default, the repository is publicly accessible. If you need a private repository, then select the **Private** check box.

Warning

Do not select `Initialize the repository with a README`. Also, do not add a `.gitignore` file nor a license to your repository.

Create an empty repository to avoid a merge conflict in a later step.

Click **Create Repository** to create the new GitHub repository. A summary page provides Git commands for a variety of project initialization scenarios:

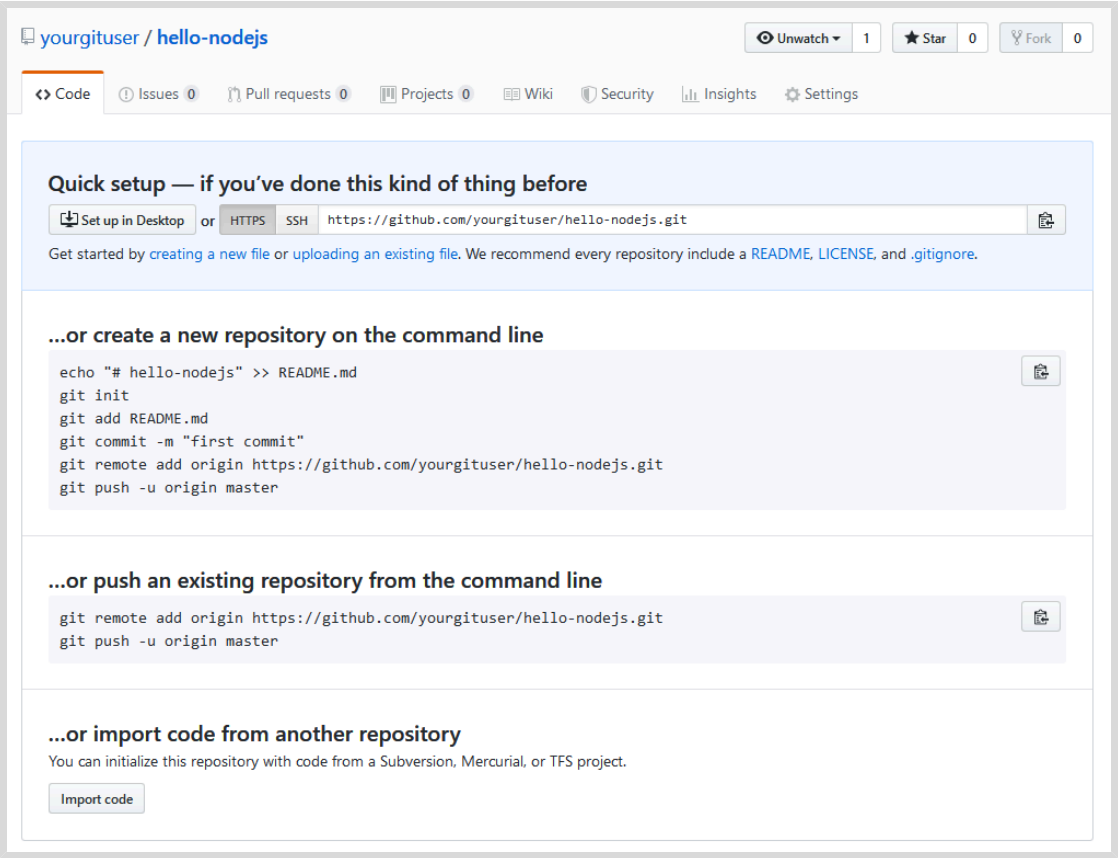


Figure 1.39: The summary page for a new GitHub repository.

7.4. Create a personal access token to allow access to your remote repository. Select your profile icon in the upper right hand corner and click (**Your Profile Icon** > **Settings**).

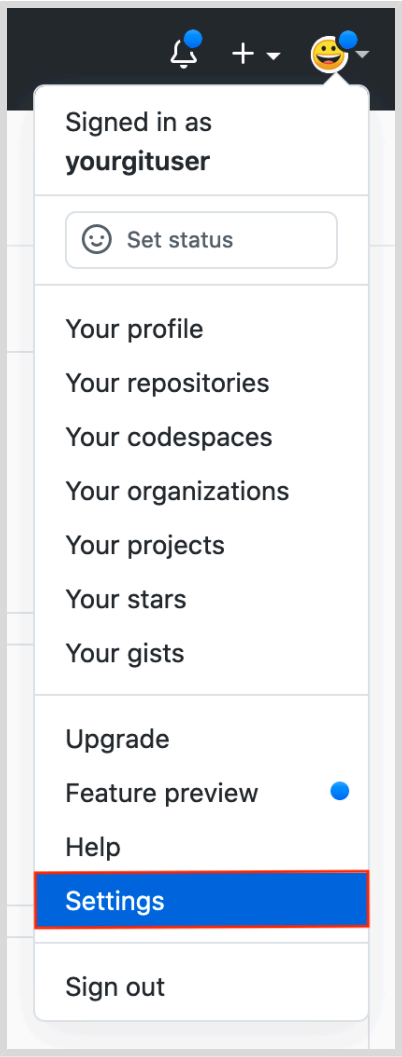


Figure 1.40: GitHub profile settings menu item.

7.5. On the left sidebar, click `Developer settings`.

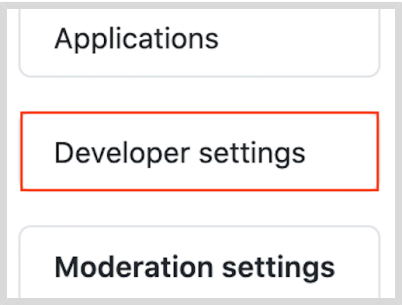


Figure 1.41: GitHub developer settings menu item.

7.6. In the left sidebar, click `Personal access tokens`.

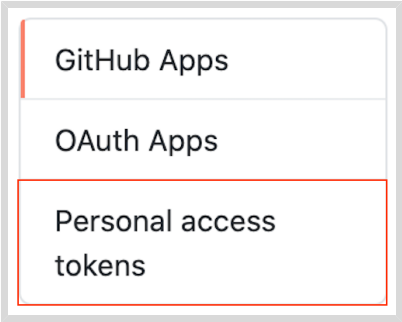


Figure 1.42: GitHub personal access token menu item.

7.7. Click `Generate new token`.

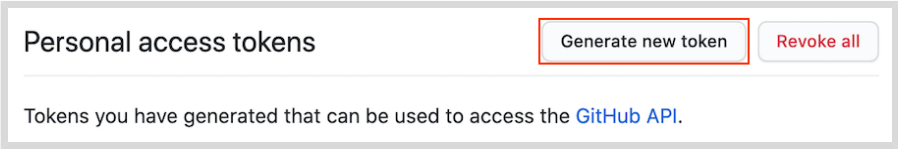


Figure 1.43: Generate new GitHub token button.

7.8. In the `Note` field, give your token a descriptive name. Select `30 days` as the `Expiration` date for you token.

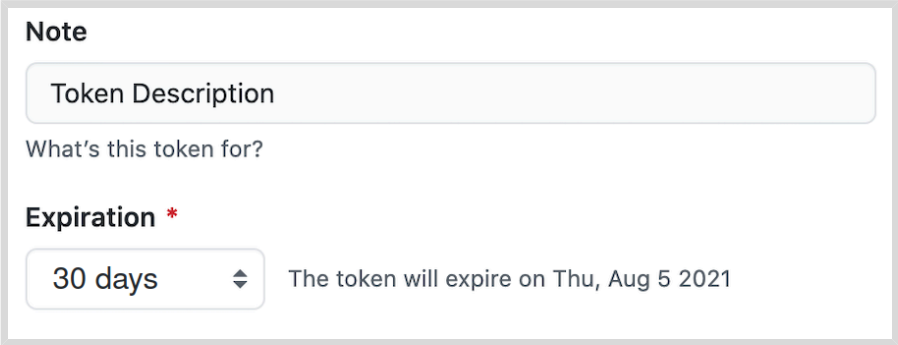


Figure 1.44: Token note and expiration fields.

7.9. Select the scopes, or permissions, you would like to grant this token. To use your token to access repositories from the command line, select `repo`.

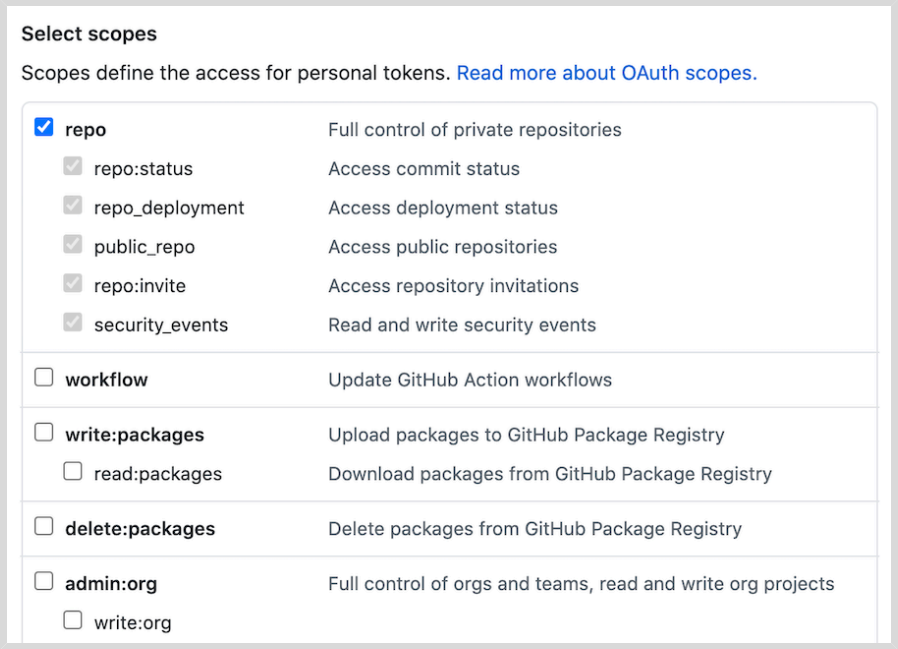


Figure 1.45: Token scope and permissions settings.

7.10. Click **Generate token**.

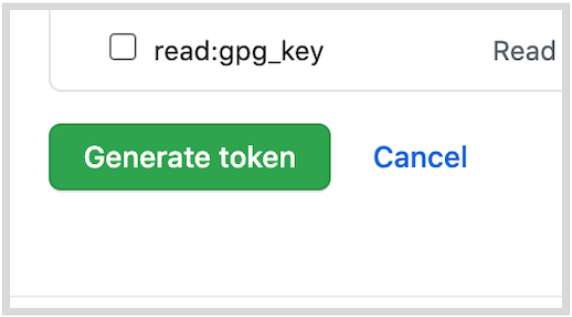


Figure 1.46: Generate the GitHub token button.

7.11. Copy the generated token.

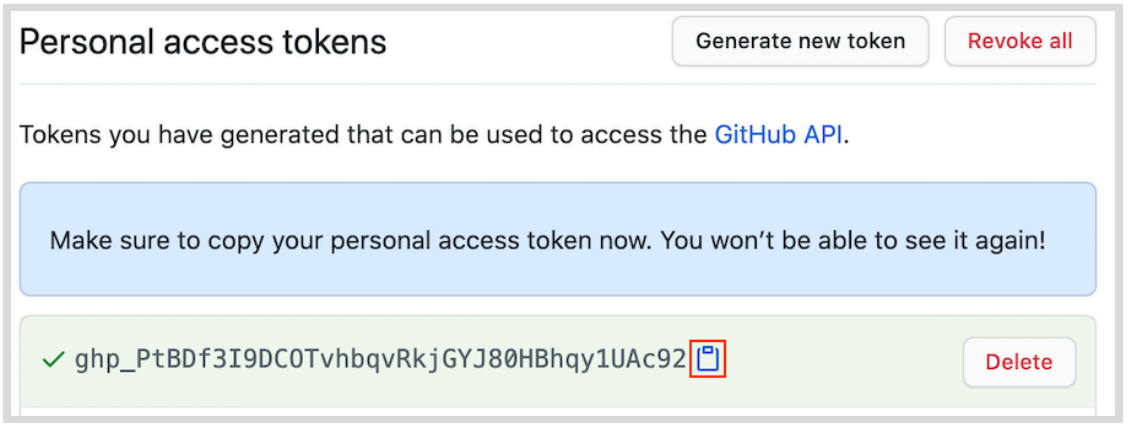


Figure 1.47: Copy the generated token to the clipboard.

7.12. Save the personal access token to be used later.

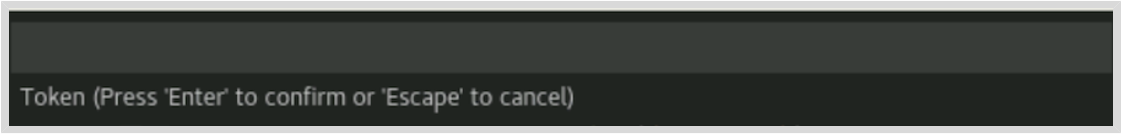


Figure 1.48: Use the generated token.

8. Add your new GitHub repository as a remote repository for the `hello-nodejs` project.

8.1. In VS Code, type `Git: Add` in the Command Palette (**View** > **Command Palette...**). Then, select `Git: Add Remote` from the list of options.

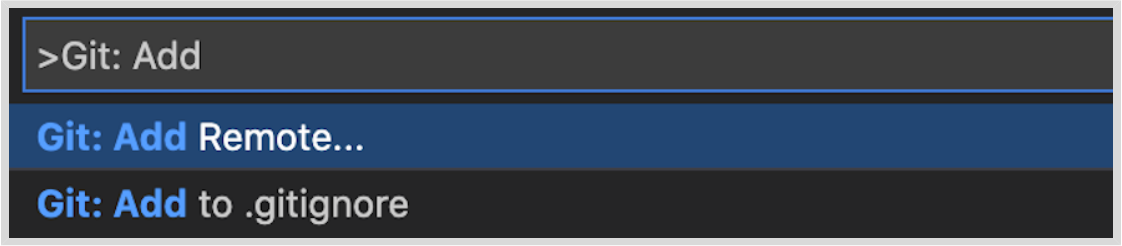


Figure 1.49: The remote URL prompt to add a remote Git repository.

8.2. If you have more that one local Git repository in VS Code, then select `hello-nodejs` from the list of options.

8.3. At the next prompt, enter the HTTPS URL of your `hello-nodejs` GitHub repository with the personal access token. The URL form is: `https://`*`TOKEN`*`@github.com/`*`yourgituser`*`/hello-nodejs`.

In the URL, replace *TOKEN* with the generated personal access token.

Warning
If you do not add the personal access token to your remote URL, you will not be authorized to push commits to your repository.

When prompted for a remote name, enter `origin`.

Note
A Git repository can interact with multiple remote repositories. A remote name of `origin` is a Git convention to indicate the originating repository for a local Git repository.

9. Publish your local repository commits to the GitHub repository.

9.1. Locate the `hello-nodejs` entry in the `SOURCE CONTROL` section, and then click the "Publish Changes" icon.

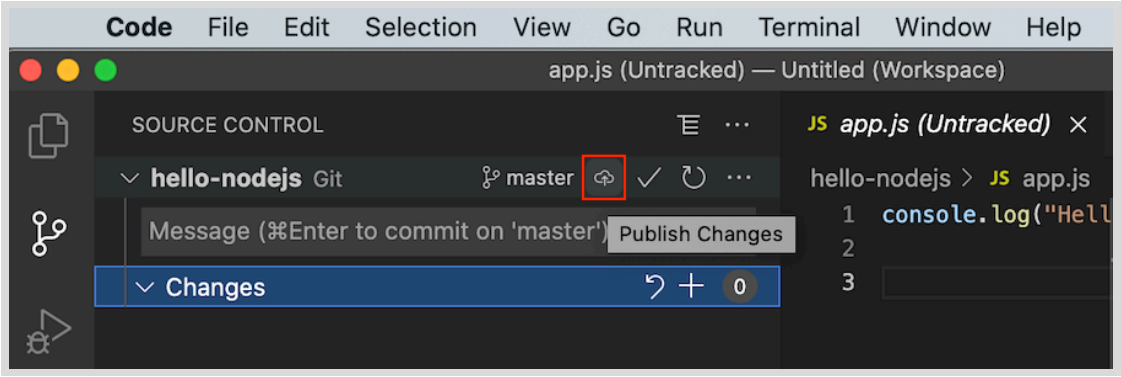


Figure 1.50: Source control view with publish changes highlighted.

The first time VS Code connects to GitHub, a prompt for your GitHub credentials displays. When prompted, provide your GitHub personal access token.

9.2. If this is your first time publishing commits in VS Code, then an additional prompt displays:

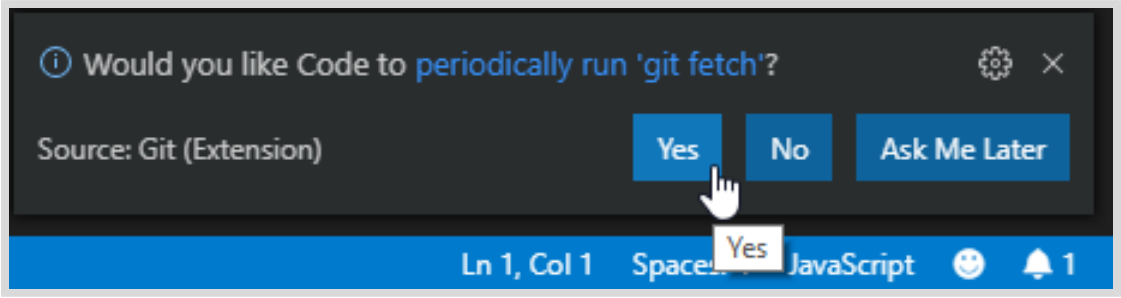


Figure 1.51: VS Code prompt to periodically fetch new commits.

Click **Yes** to configure VS Code to periodically check the remote repository for new commits.

10. In a browser, navigate to `https://github.com/`*`yourgituser`*`/hello-nodejs`, replacing *`yourgituser`* with your GitHub user name. Verify that your source code is present in your GitHub repository.

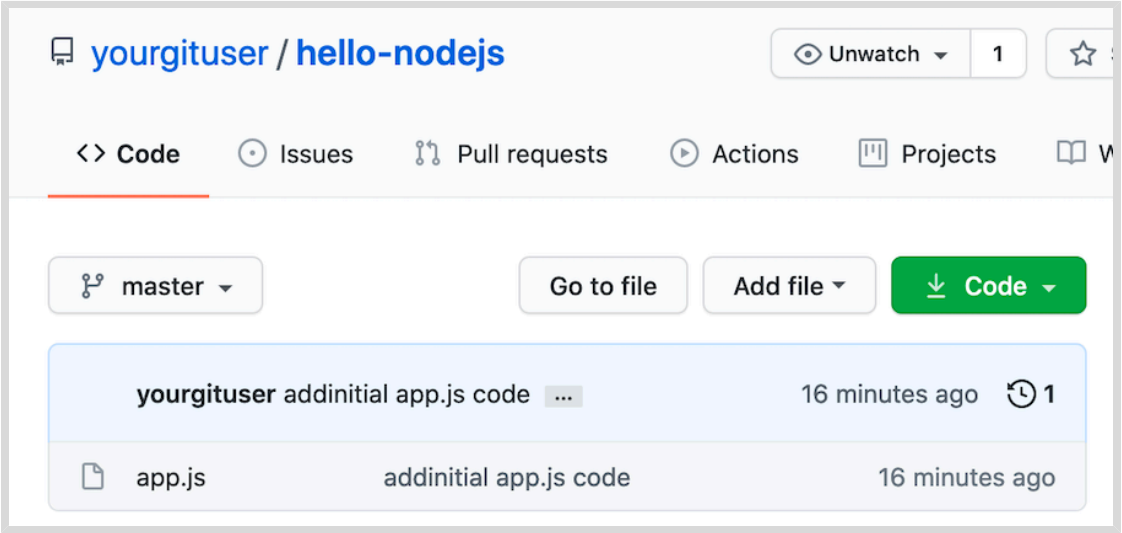


Figure 1.52: Local project files are present on GitHub.

11. To clean up your work, click the **Kill Terminal** icon to close the integrated terminal window.

Finish

This concludes the guided exercise.

Managing Application Source Code with Git

Objectives

After completing this section, you should be able to use version control to collaborate and manage application source code.

Overview of Git Branching

Git version control features a branching model to track code changes. A **branch** is a named reference to a particular sequence of commits.

All Git repositories have a base branch. A base branch named `main` is preferred by many projects. Some platforms use `master` by default. When you create a commit in your repository, the base branch is updated with the new commit.

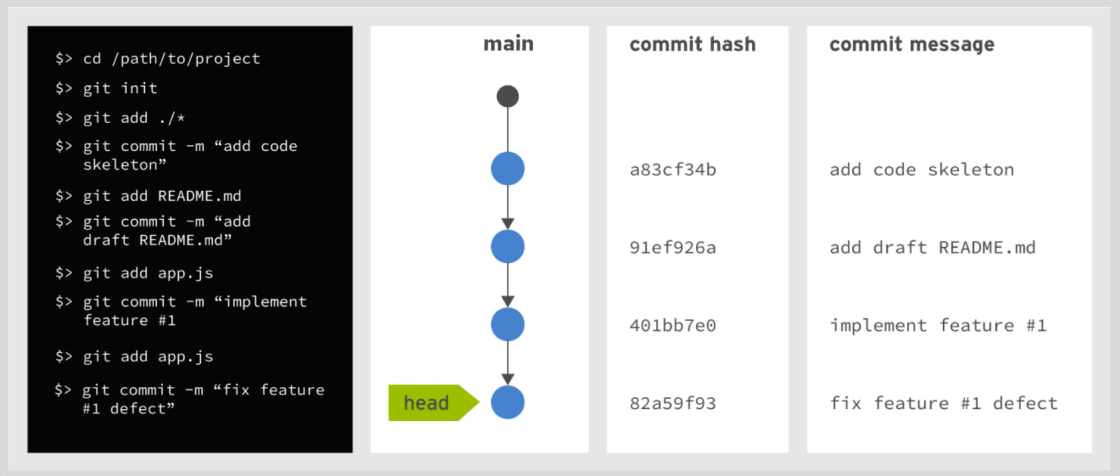


Figure 1.53: Commits to the main branch of a Git repository.

By convention, the `main` branch in a Git repository contains the latest, stable version of the application source code. To implement a new feature or functionality, create a new branch from the `main` branch. This new branch, called a **feature branch**, contains commits corresponding to code changes for the new feature. The `main` branch is not affected by commits to the feature branch.

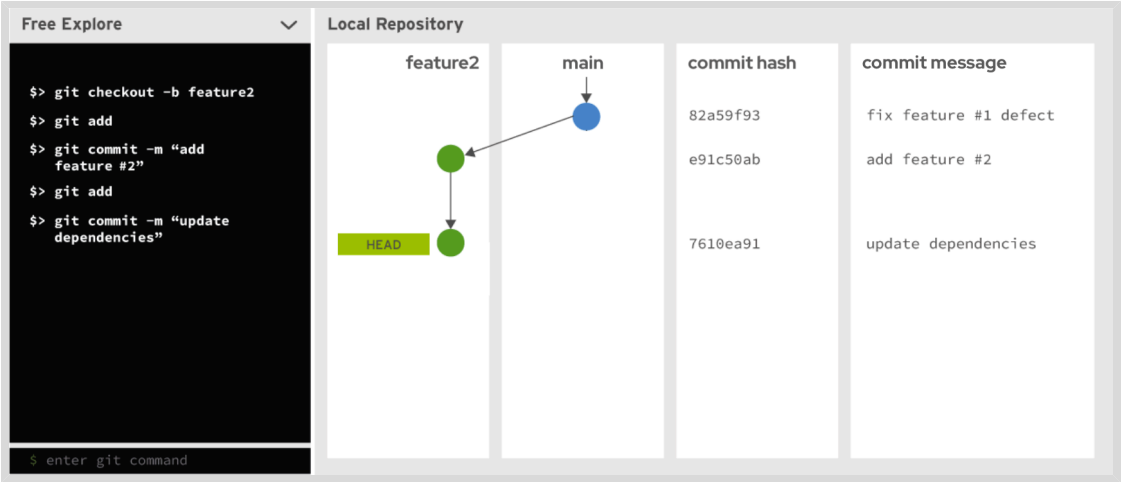


Figure 1.54: Commits to a feature branch of a Git repository.

When you use a branch for feature development, you can commit and share your code frequently without impacting the stability of code in the `main` branch. After ensuring the code in the feature branch is complete, tested, and reviewed, you are ready to merge the branch into another branch, such as the `main` branch. **Merging** is the process of combining the commit histories from two separate branches into a single branch.

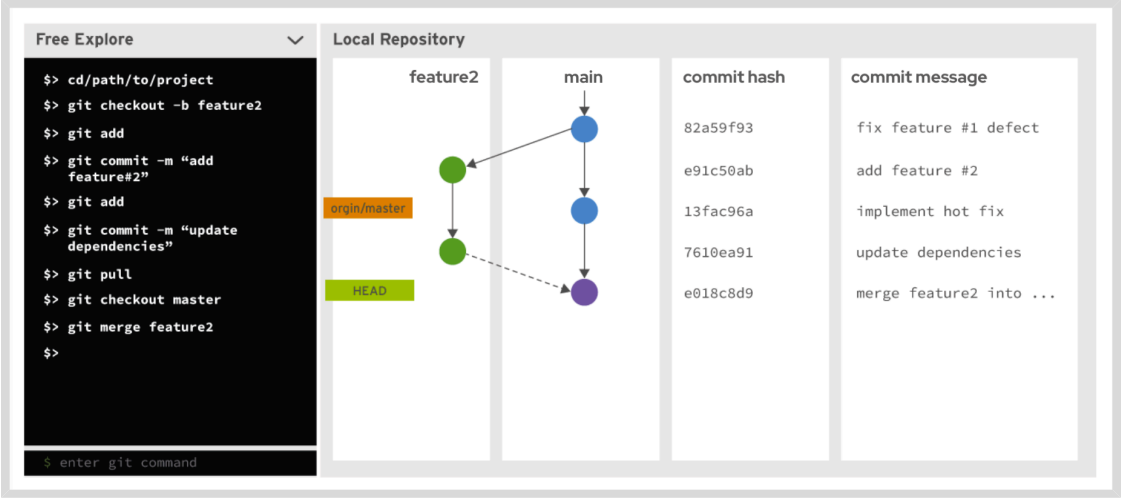


Figure 1.55: Merging a feature branch into the main branch.

Merge Conflicts

Git has sophisticated mechanisms to merge code changes from one branch into another branch. However, if there are changes to the same file in both branches, then a **merge conflict** can occur.

A merge conflict indicates that Git is not able to automatically determine how to integrate changes from both branches. When this happens, Git labels each affected change as a conflict. VS Code displays an entry for each file with a merge conflict in the Source Control view, beneath the `MERGE CHANGES` heading. Each file with a merge conflict entry contains a `C` to the right of the entry.

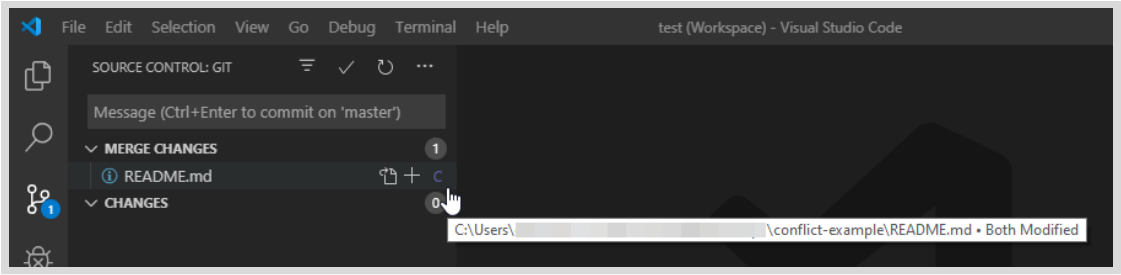


Figure 1.56: A merge conflict in the Source Control view of VS Code.

Git also inserts markers in each affected file to indicate the sections that contain content conflicts from both branches. If you click the merge conflict entry in the VS Code Source Control view, then an editor tab displays and highlights the sections of the file that conflict.

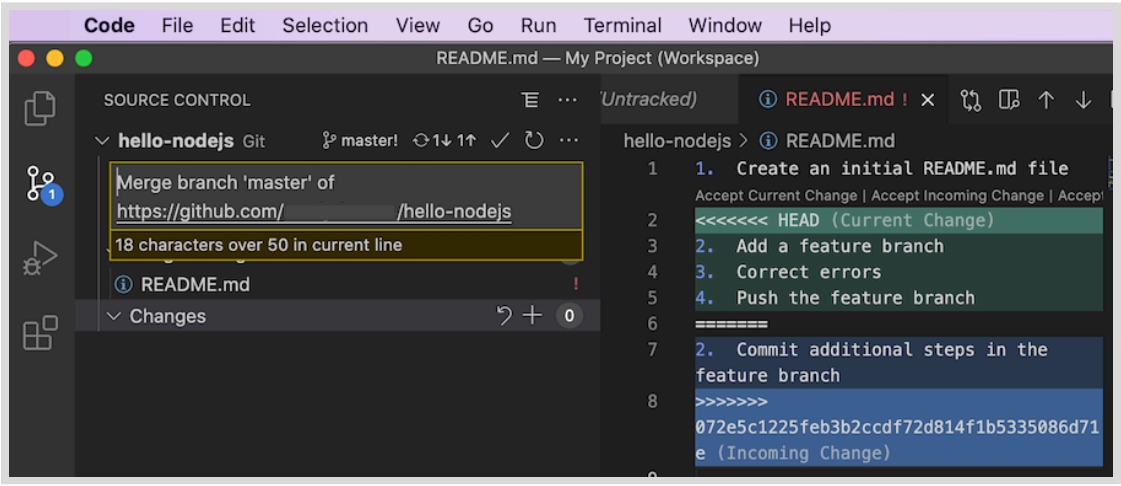


Figure 1.57: VS Code highlights the conflicts in a merge conflict file.

In the preceding figure, the green highlighted section contains content from the current branch, in this case the `master` branch. The blue highlighted text contains incoming changes pulled in from the remote repository, which is being merged into the `master` branch.

There are many options to resolve merge conflicts that happen as a result of the normal merging process.

For each conflict in a conflicted file, replace all of the content between the merge conflict markers (<<<<<<<< and >>>>>>>> inclusive) with the correct content from one branch or the other, or an appropriate combination of content from both branches.

For example, the following figure shows that the conflict section from the preceding example is replaced with content from both the `master` and `update-readme` branches.

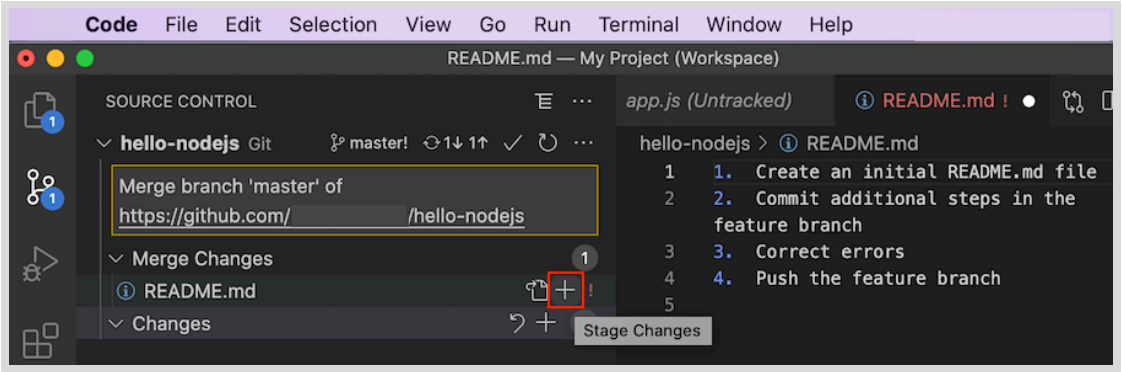


Figure 1.58: Resolving a merge conflict in VS Code.

After you reconcile the content for each conflict in a file, then save, stage, and commit the file changes.

> **Note**
> Managing merge conflicts is beyond the scope of this course. For more information on merge conflicts, see **Basic Merge Conflicts** at https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Collaboration Strategies Using Git

Git provides development teams with the flexibility to implement code collaboration processes and branching strategies that fit the size and dynamics of each team. When a team agrees on a Git workflow, then code changes are managed consistently and team productivity increases.

Centralized Workflow

A centralized Git workflow uses a central Git repository as the single source of record for application code. The central repository is typically hosted on a repository hosting platform, such as GitHub. In this workflow, the development team agrees that the central repository will only contain a single branch, `main`. Developers push code changes directly to the `main` branch, and do not push commits in other branches to the central repository.

Developers clone the central repository, creating a local copy of the repository for each developer. Developers make code changes and commit to the `main` branch of their local repository. Developers merge in the latest changes from the remote repository, before pushing new changes to the remote repository.

Because the workflow results in commits to a single branch, conflicts occur often. Merge conflicts are mitigated when teams clearly identify separate code changes for each team member that can be implemented in distinct files.

In addition to merge conflicts, this workflow allows you to commit partial or incomplete code changes to the `main` branch. Incomplete code compromises the stability of the `main` branch. Teams that use a centralized workflow must adopt additional processes and communication strategies to mitigate these risks.

One way teams mitigate code stability issues with a centralized worfklow is to implement Git tags in the code repository. A **tag** is a human-readable reference to a particular commit in a Git repository, independent of any branch. Teams can implement a tag nomenclature to identify commits corresponding to stable versions of the project source code.

Note

Git tags are beyond the scope of this course. See **Tagging** in the Git documentation at https://git-scm.com/book/en/v2/Git-Basics-Tagging.

The centralized Git workflow works well for small teams collaborating on a small project with infrequent code changes, but becomes difficult to manage with larger teams and large code projects.

Feature Branch Workflow

A feature branch workflow implements safety mechanisms to protect the stability of code on the `main` branch. The aim of this workflow is to always have deployable and stable code for every commit on the `main` branch, but still allow team members to develop and contribute new features to the project.

In a feature branch workflow, each new feature is implemented in a dedicated branch. One or more contributors collaborate on the feature by committing code to the feature branch. After a feature is complete, tested, and reviewed in the feature branch, then the feature is merged into the `main` branch.

The feature branch workflow is an extension of the centralized workflow. A central repository is the source of record for all project files, including feature branches.

When a developer is ready to merge a feature branch into the `main` branch, the developer pushes the local feature branch to the remote repository. Then, the developer submits a request to the team, such as a pull request or merge request, to have the code changes in the feature branch reviewed by a team member. After a team member approves the request, the repository hosting platform merges the feature branch into the `main` branch.

A **pull request** is a feature of several repository hosting platforms, including GitHub and BitBucket. A pull request lets you submit code for inclusion to a project code base. Pull requests often provide a way for team members to provide comments, questions, and suggestions about submitted code changes. Pull requests also provide a mechanism to approve the merging of the code changes into another branch, such as `main`.

Note

On other platforms, such as GitLab and SourceForge, this code review feature is called a **merge request**.

For example, the following figure shows the GitHub user interface after pushing the `feature1` branch. GitHub displays a notification that you recently pushed the `feature1` branch. To submit a pull request for the `feature1` branch, click **Compare & pull request** in the notification.

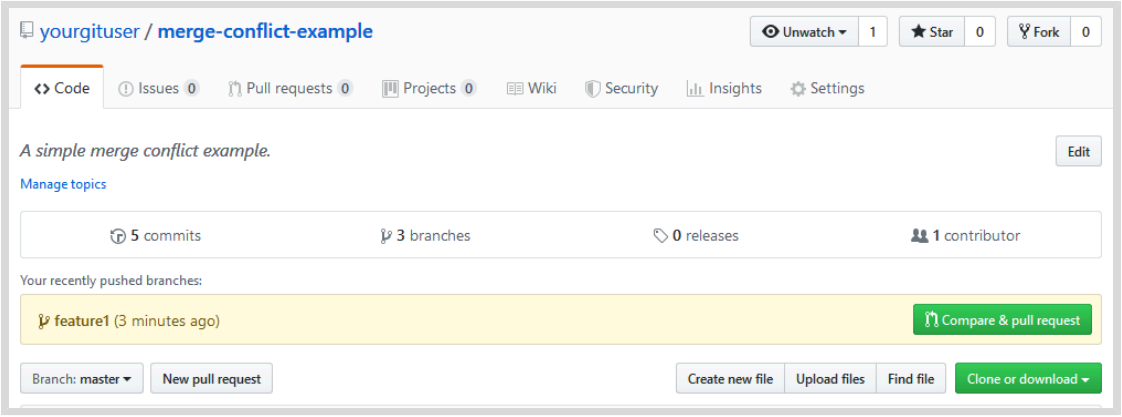


Figure 1.59: GitHub user inteface after you push a feature branch.

GitHub displays a form that you must complete to submit the pull request. The GitHub pull request form allows you to provide a title and description for the pull request. The form also allows you to assign the pull request to a GitHub user, and also specify a set of GitHub users to review the code changes in the pull request.

By default, you request that the `feature1` branch be merged (or pulled) into the `main` branch. If you want to merge the feature branch into a different branch, then you select the correct base branch from the list of options.

GitHub displays a message to indicate if the merge operation will cause conflicts. In the figure that follows, GitHub indicates that the two branches will merge without any conflicts and can be automatically merged.

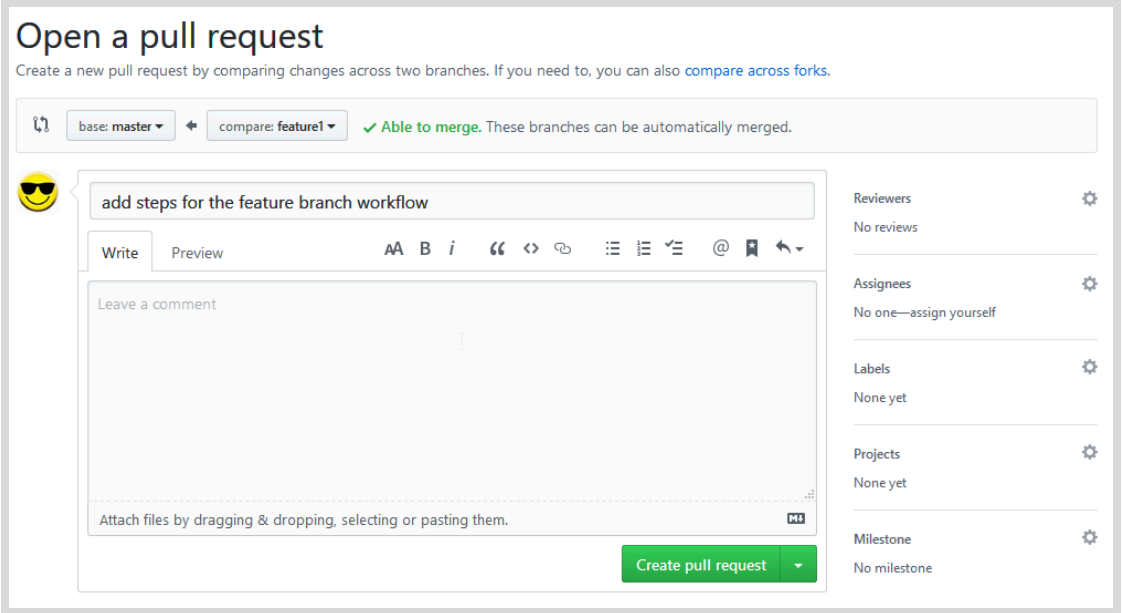


Figure 1.60: The GitHub form to submit a pull request.

After completing the form, click **Create pull request** to create the pull request.

Click **Pull requests** for the GitHub repository to display a list of pull requests. Select your pull request from the list. GitHub displays information about the pull request including comments, questions, or suggestions by other reviewers.

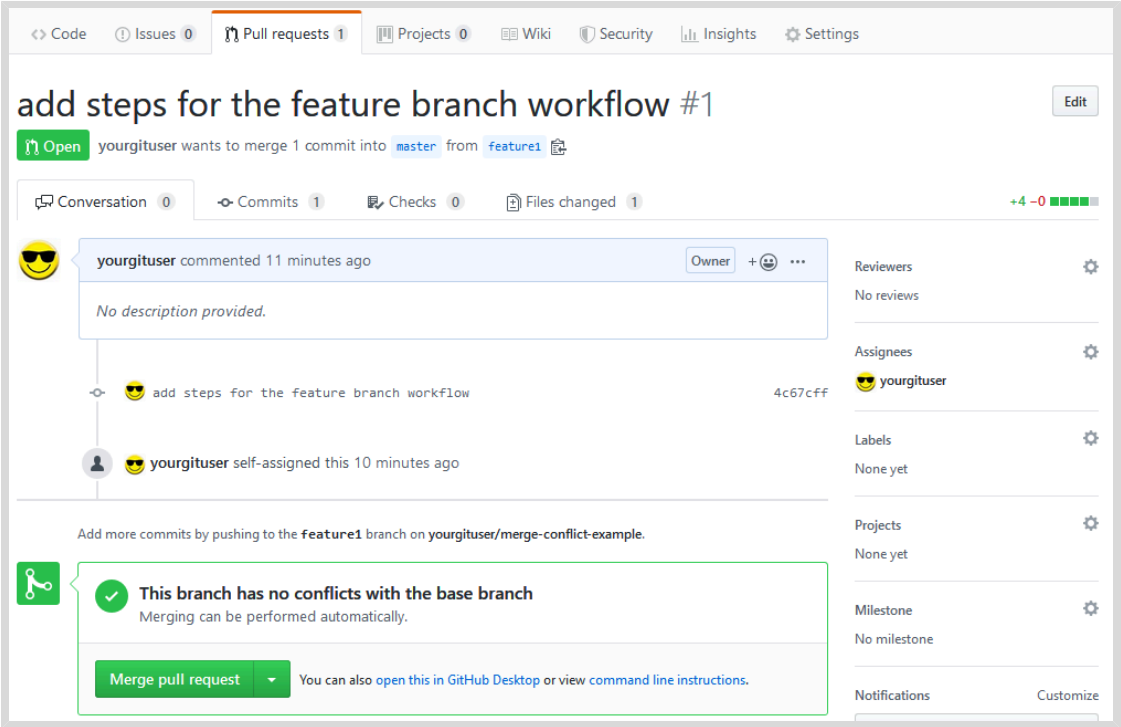


Figure 1.61: A GitHub pull request.

After a reviewer grants approval, you can merge the pull request. When you are ready to merge the changes to the `main` branch, click **Merge pull request**. GitHub displays a message for the merge operation. Click **Confirm merge** to merge the `feature1` branch into the `main` branch.

Finally, in the feature branch workflow you delete a feature branch after it is merged into the `main` branch. Click **Delete branch** to delete the branch.

Gitflow Workflow

The `Gitflow Workflow` uses all the same constructs discussed in the `Feature Branch Workflow` and also defines a strict branching model around project releases. Gitflow is ideally suited for projects that have a scheduled release cycle and for the DevOps best practice of continuous delivery. In addition to feature branches, it uses individual branches for preparing, maintaining, and recording releases.

Instead of a single `main` branch, this workflow uses two branches to record the history of the project. The `main` branch stores the official release history, and the `develop` branch serves as an integration branch for features. It's also convenient to tag all commits in the `main` branch with a version number.

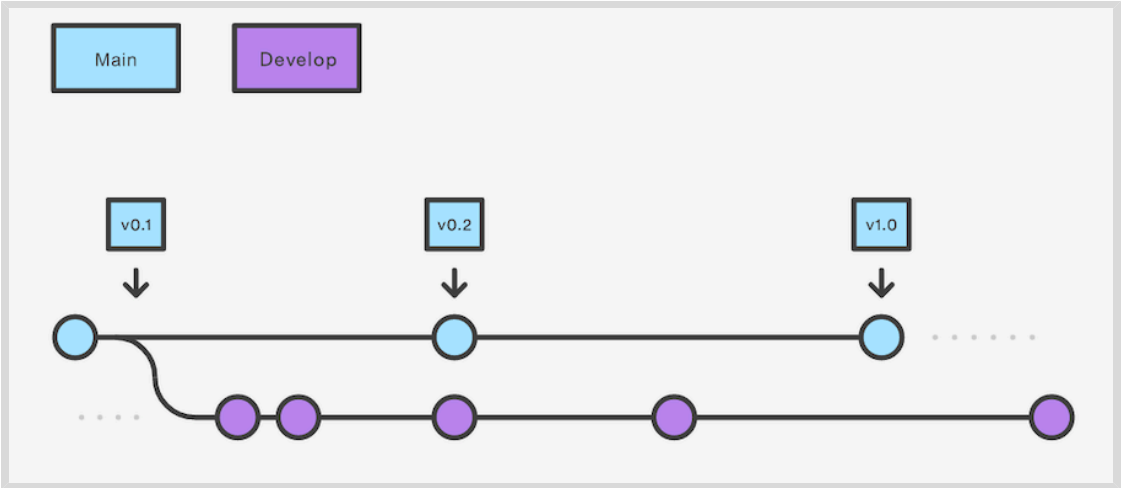


Figure 1.62: Gitflow develop and main branches.

Instead of branching off of the `main`, feature branches use the `develop` branch as their parent. Completed features are merged back into `develop` and never interact with `main`.

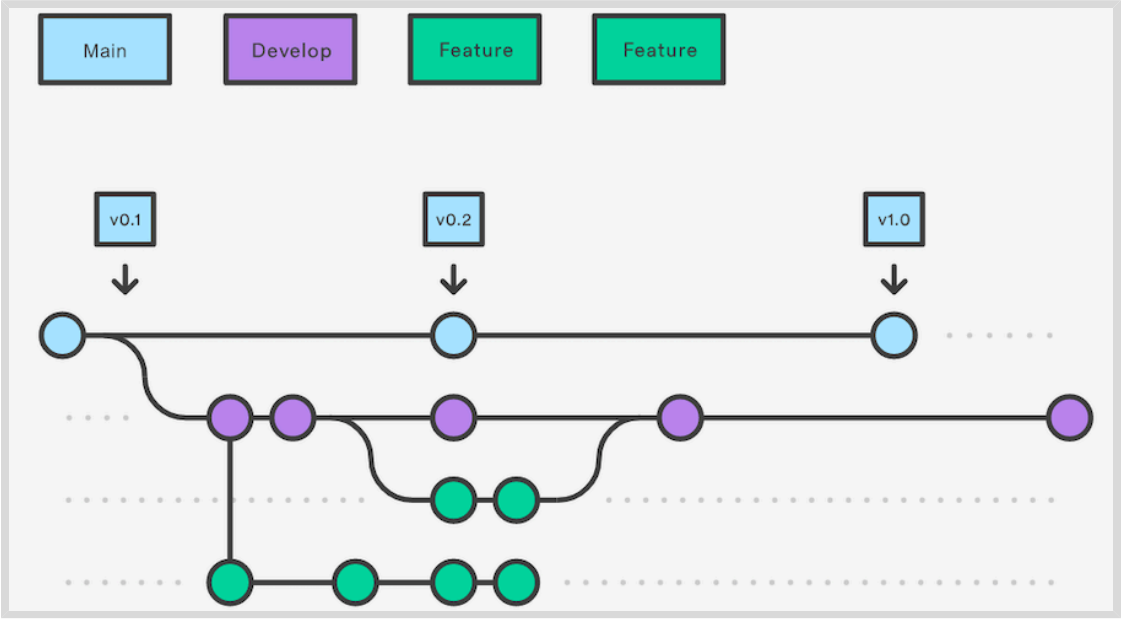


Figure 1.63: Gitflow feature branches from develop.

Once the `develop` branch has acquired enough features, a release branch is created with `develop` as the parent. Creating this branch starts the next release cycle, so no new features can be added after this point. Only bug fixes, documentation, or other release-oriented assets should be added. Once it is ready to ship, the release branch gets merged into `main` and tagged with a version number.

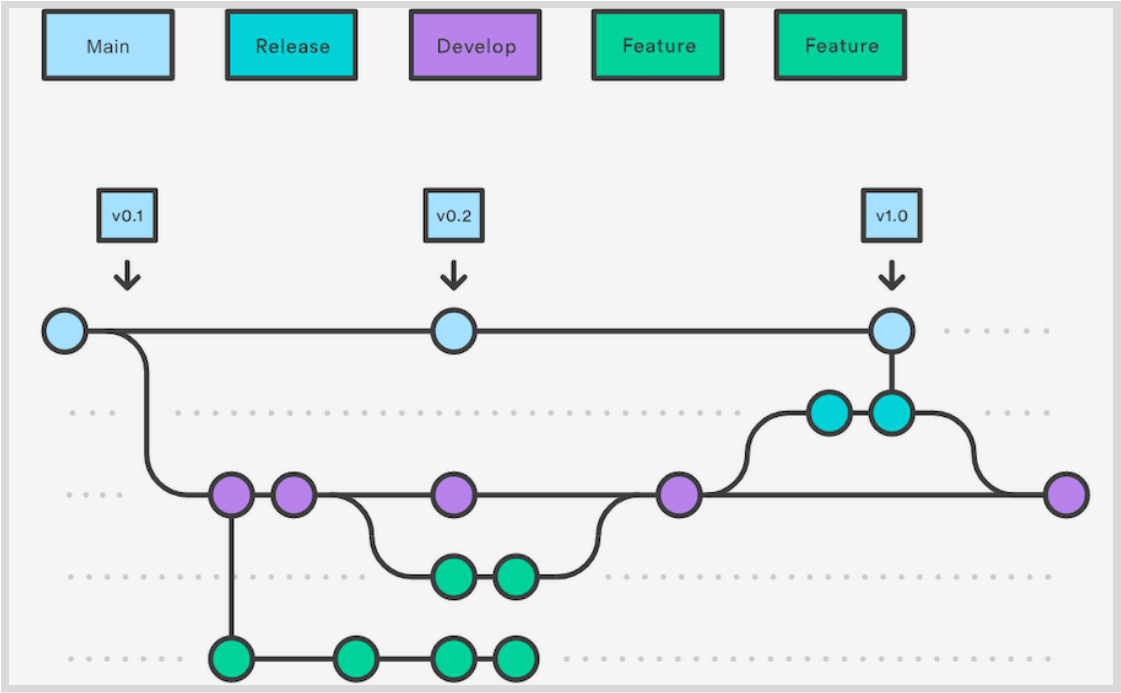


Figure 1.64: Gitflow release branches.

Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release. It also creates well-defined phases of development that are seen in the structure of the repository.

Forked Repository Workflow

The Forked repository workflow is often used with large open source projects. With a large number of contributors, managing feature branches in a central repository is difficult.

Additionally, the project owner may not want to allow contributors to create branches in the code repository. In this scenario, branch creation on the central repository is limited to a small number of team members. The forked repository workflow allows a large number of developers to contribute to the project while maintaining the stability of the project code.

In a forked repository workflow, you create a **fork** of the central repository, which becomes your personal copy of the repository on the same hosting platform. After creating a fork of the repository, you clone the fork. Then, use the feature branch workflow to implement code changes and push a new feature branch to your fork of the official repository.

Next, open a pull request for the new feature branch in your fork. After a representative of the official repository approves the pull request, the feature branch from your fork is merged into the original repository.

For example, consider the workflow of the OpenShift Origin GitHub repository.

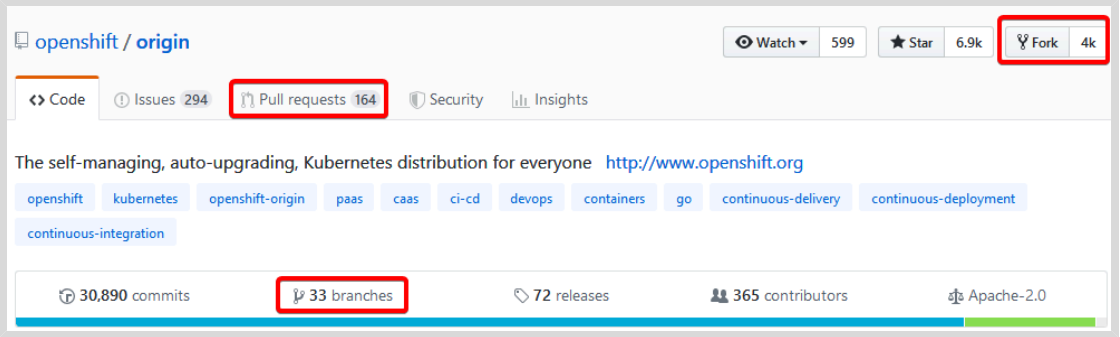


Figure 1.65: The OpenShift Origin GitHub repository.

Notice that there are more pull requests than branches in the OpenShift Origin GitHub repository. Most of the pull requests contain code from one of the forks of the repository, instead of from a repository branch.

Git Branching in VS Code

In VS Code, use the Source Control view (**View** > **SCM**) to access the branching features in Git. The `SOURCE CONTROL REPOSITORIES` section contains an entry for each Git repository in your VS Code workspace. Each entry displays the current branch for the repository.

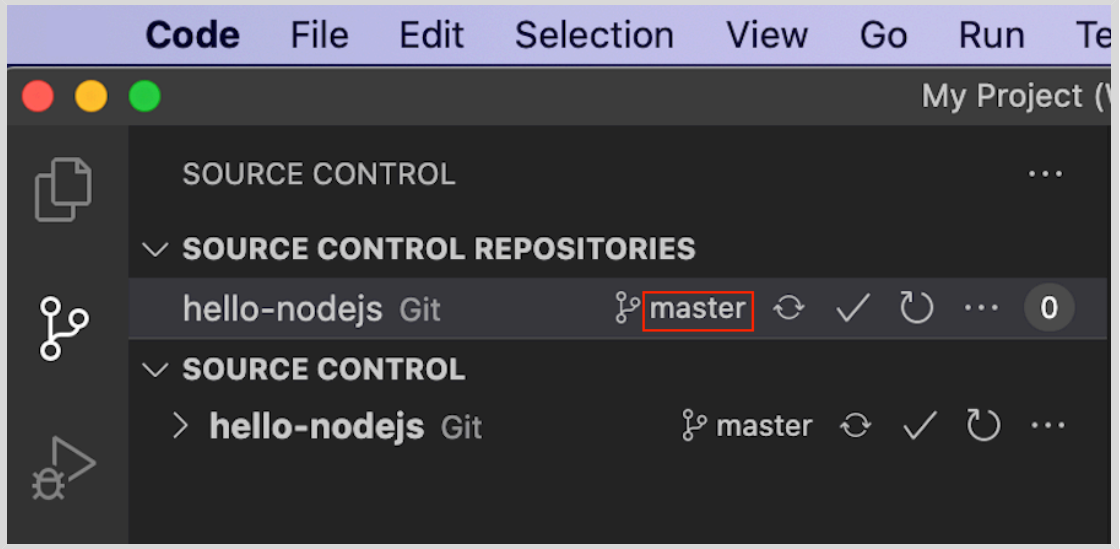


Figure 1.66: The Source Control view shows the current branch for each workspace Git repository.

To switch to a different branch in a repository, click the current branch name for the repository entry under the `SOURCE CONTROL REPOSITORIES` heading. VS Code displays two options to create a new branch, and also displays a list of existing branches and tags in the repository:

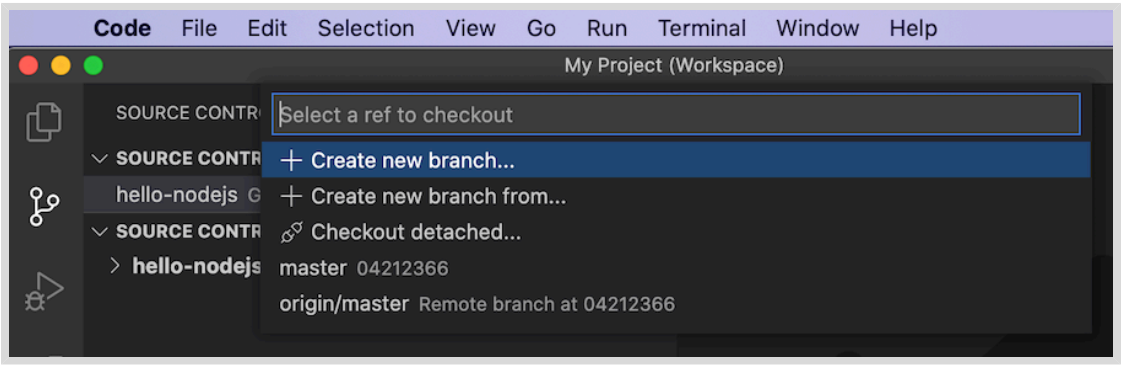


Figure 1.67: Checkout a branch in the Source Control view.

When you select either of the two options to create a new branch, VS Code prompts you for a name to assign to the new branch. If you selected the `Create new branch...` option, VS Code creates a new branch from the current repository branch.

If you selected the `Create a new branch from...` option, then VS Code also provides you list of existing repository tags and branches. After you select an item, VS Code creates a new branch that begins from the tag or branch that you select.

Note

Often software projects adopt branch naming conventions or standards. Branch naming standards help you summarize the code changes contained in a branch.

The following are examples of branch name templates for a branch naming standard:

- `feature/`*`feature-id/description`*
- `hotfix/`*`issue-number/description`*
- `release/`*`release-string`*

A branch naming standard also defines the set of allowable characters. Branch names are often limited to alphanumeric characters and field seperators (such as `/`, `_`, or `-` characters).

After VS Code creates the new local branch, the Source Control view updates the repository entry with the name of the new branch. When you click the `Publish Changes` icon, VS Code publishes the new local branch to the remote repository.

Any new code you commit is added to the new local branch. When you are ready to push your local commits to the remote repository, click the `Synchronize Changes` icon in the Source Control view. The `Synchronize Changes` icon displays:

- the number of commits in the local repository to upload
- the number of commits in the remote repository to download

To download commits from a remote repository, VS Code first fetches the remote commits and merges them into your local repository. Then, VS Code pushes your local commits to the remote repository.

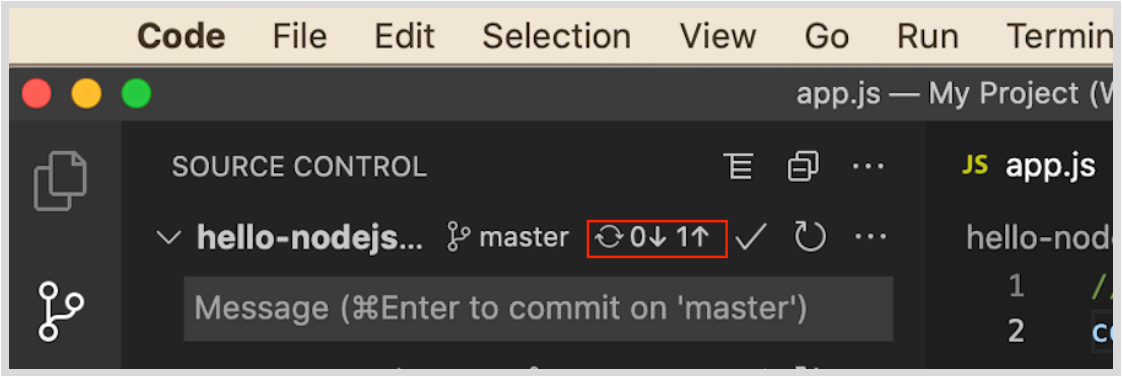


Figure 1.68: Push a local commit to the remote repository.

References

For more information about Git branches, refer to the Git documentation at
https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell

For more information about merging, refer to the Git documentation at
https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

For more information about using Git and version control in VS Code, refer to the VS Code documentation at
https://code.visualstudio.com/docs/editor/versioncontrol

For more information about using the Gitflow Workflow, refer to the Atlassian documenation at
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Guided Exercise

Managing Application Source Code with Git

In this exercise, you will use VS Code to push code changes in a new branch to a remote Git repository.

Outcomes

You should be able to:

- Fork the sample applications repository for this course to your personal GitHub account.
- Clone the sample applications repository for this course from your personal GitHub account to your system.
- Commit code changes to a new branch.
- Push a new branch to a remote repository.

Before You Begin

To perform this exercise, ensure that:

- Visual Studio Code (VS Code) is installed on your system.
- Node.js and Node Package Manager (NPM) are installed on your system.
- Git is installed on your system and configured with your user name and email address.

Instructions

1. Fork the sample applications for this course into your personal GitHub account.

1.1. Open a web browser, and navigate to https://github.com/RedHatTraining/DO101x-apps. If you are not logged in to GitHub, then click **Sign in** in the upper-right corner.

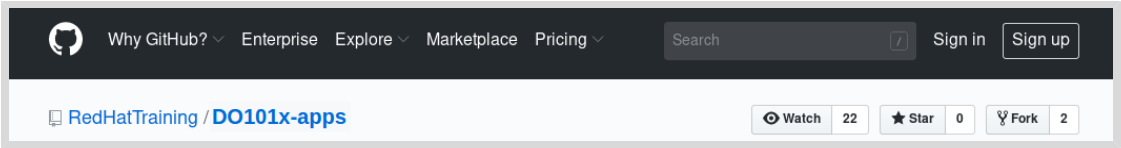


Figure 1.69: Sign-in page for the course Git repository.

1.2. Log in to GitHub using your personal user name and password.

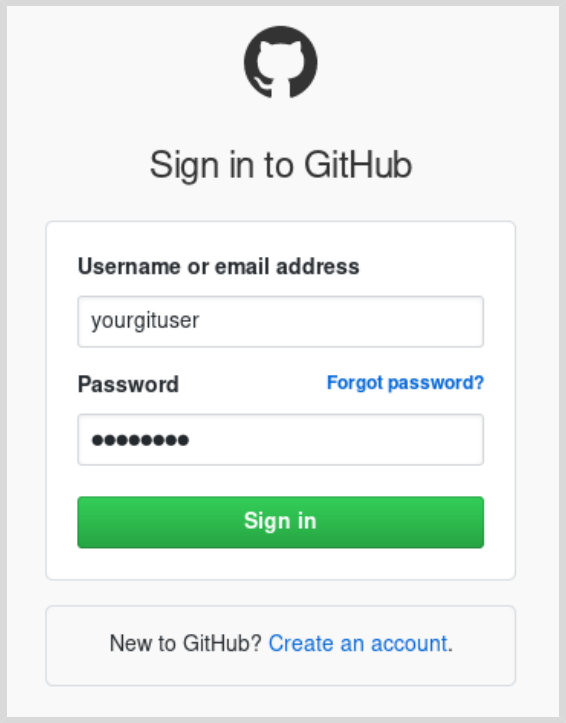


Figure 1.70: GitHub Sign-in page.

1.3. Return to https://github.com/RedHatTraining/DO101x-apps and click **Fork** in the upper-right corner.

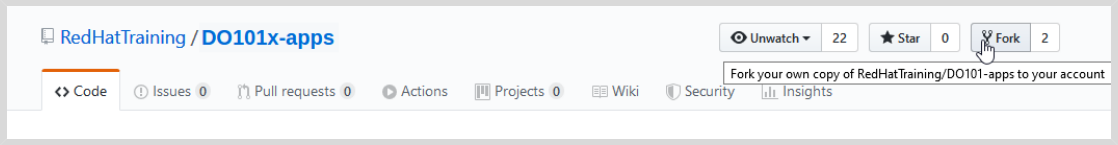


Figure 1.71: Fork the Red Hat Training DO101x-apps repository.

1.4. In the `Fork DO101x-apps` window, click *`yourgituser`* to select your personal GitHub project.

Important

While it is possible to rename your personal fork of the https://github.com/RedHatTraining/DO101x-apps repository, the example output in this course assumes that you retain the name `DO101x-apps` when you fork the repository.

1.5. After a few minutes, the GitHub web interface displays your new *`yourgituser`*`/DO101x-apps` repository.

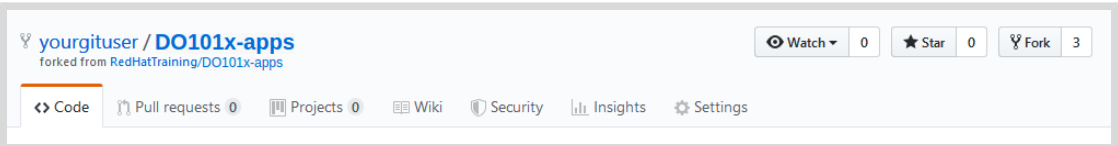


Figure 1.72: A personal fork of the Red Hat Training DO101x-apps repository.

2. Clone the sample applications for this course from your personal GitHub account using VS Code.

2.1. If you do not have VS Code open from a previous exercise, the open it.

2.2. In the Command Palette (**View** > **Command Palette...**), type `clone`. Select **Git: Clone** from the list of options.

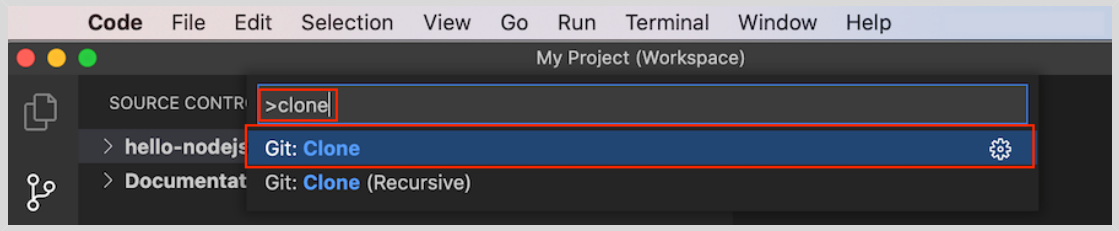


Figure 1.73: Using the command palette to clone a repository.

2.3. In the prompt that displays, enter the HTTPS URL for your repository, `https://TOKEN@github.com/yourgituser/DO101x-apps`.

In the URL, replace *TOKEN* with the generated personal access token.

2.4. In the file window that opens, choose a local folder to store the repository clone. VS Code creates a `DO101x-apps` subfolder in the folder you select.

Click **Select Repository Location** to select your location.

2.5. After VS Code clones the repository, a prompt displays in the lower-right corner of the VS Code window. Click **Add to Workspace** to add the cloned repository to your VS Code workspace.

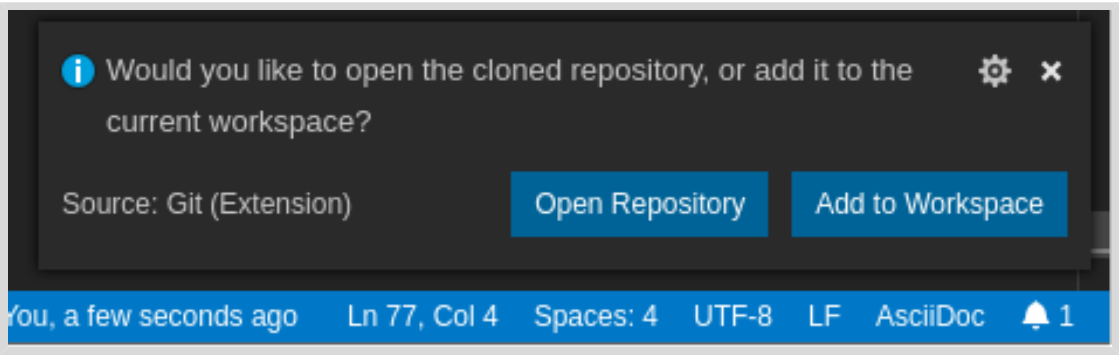


Figure 1.74: Add a cloned repository to the current workspace.

Note
This prompt remains active for only a few seconds. If the prompt closes, add the cloned repository folder to your workspace (**File** > **Add Folder to Workspace...**). In the file window that displays, navigate to the location of the cloned `DO101x-apps` folder. Select the `DO101x-apps` folder, and then click **Add**.

3. Add a feature to the `express-helloworld` application to display `Hello Mars!` when a user accesses the `/mars` application endpoint. Implement the new feature in the `devenv-versioning` branch.

3.1. From the Source Control view (**View** > **SCM**), click **main** in the `DO101x-apps` entry under `SOURCE CONTROL REPOSITORIES`.

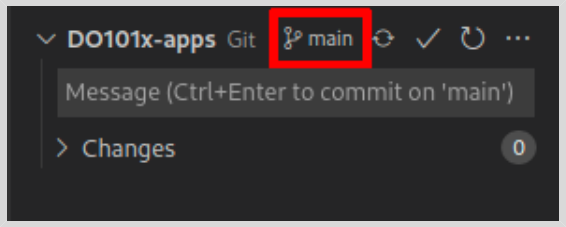


Figure 1.75: Checkout a new branch for a repository.

Note

If the Source Control view does not display the `SOURCE CONTROL REPOSITORIES` heading, right-click `SOURCE CONTROL` at the top of the Source Control view and select `Source Control Repositories`.

3.2. Select `Create new branch...` from the list of options.

3.3. When prompted, enter `devenv-versioning` for the branch name. The `DO101x-apps` entry in the Source Control view updates to the `devenv-versioning` branch.

3.4. In the Explorer view (**View** > **Explorer**), click the `DO101x-apps/express-helloworld/app.js` file. VS Code opens an editor tab for the file.

3.5. To implement the new feature, add the following lines to the `app.js` file:

```
app.get('/mars', function(req, res) {
  res.send('Hello Mars!\n');
});
```

Insert these lines before the line that begins with `app.listen`:

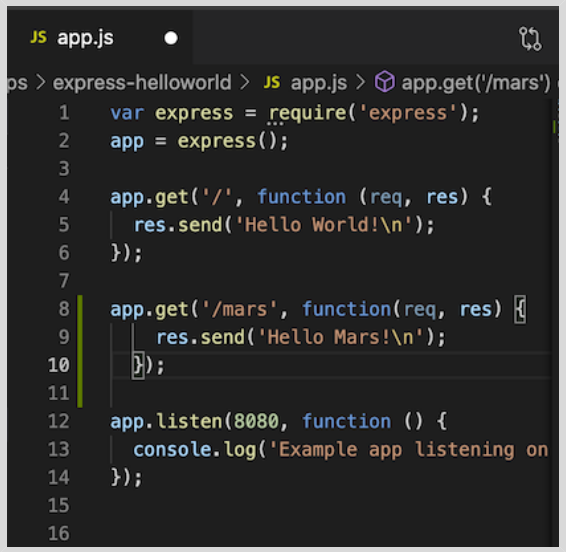


Figure 1.76: An added feature to the Express application.

Save the `app.js` file (**File** > **Save**).

4. Install the application dependencies and execute the application. Verify that the application returns a `Hello Mars!` message when you access the `/mars` endpoint.

> **Note**
> This step requires the Node Package Manager (NPM). When you install the Node.js package on Ubuntu systems, NPM is not installed as a dependency. Thus, you must also install the NPM package on Ubuntu systems.
>
> The Ubuntu NPM package installs several different software development packages. You can skip this step if you need to minimize the number of installed packages on your Ubuntu system. Skipping this step does not prevent you from completing this exercise.

4.1. Right-click `express-helloworld` in the Explorer view (**View** > **Explorer**) and then select `Open in Integrated Terminal`.

4.2. Execute the `npm install` command in the integrated terminal to install the application dependencies.

4.3. Execute the `npm start` command in the integrated terminal to start the application.

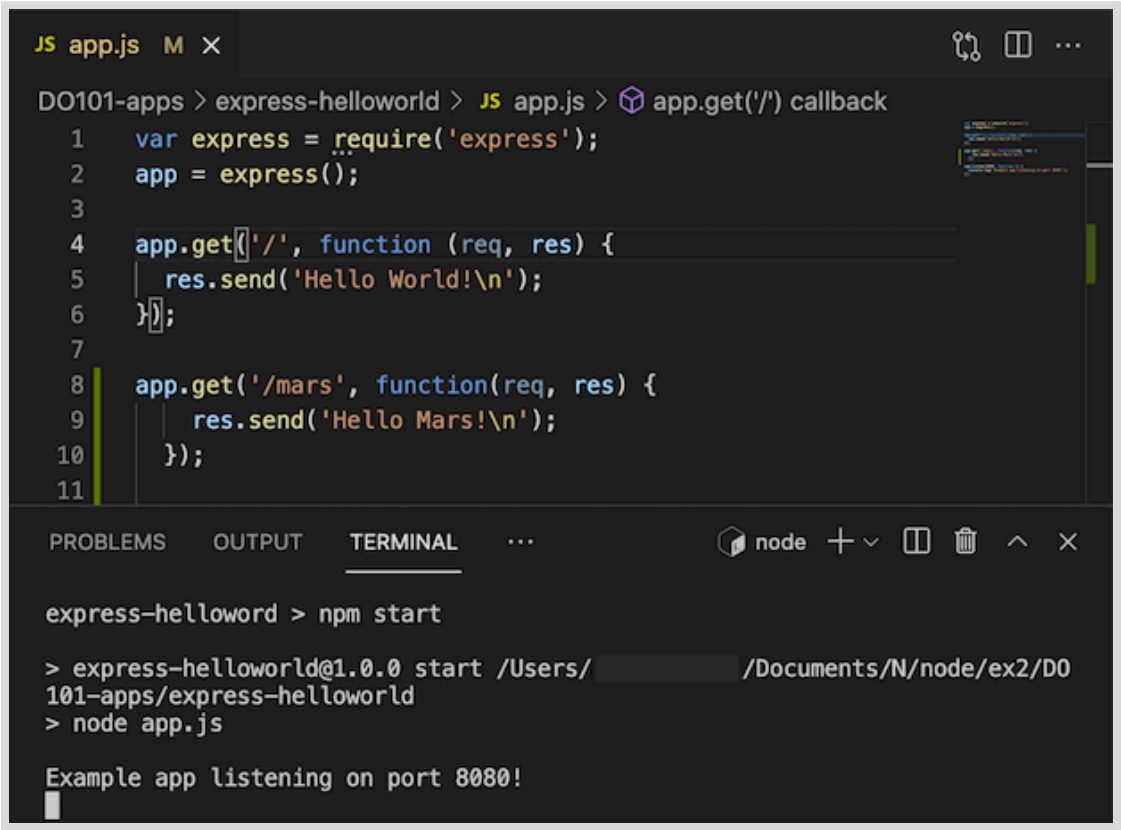


Figure 1.77: Executing a Node.js application from the integrated terminal.

4.4. In a browser, navigate to `http://localhost:8080/`. Verify that the application responds with a `Hello World!` message.

4.5. In a browser, navigate to `http://localhost:8080/mars`. Verify that the application responds with a `Hello Mars!` message.

4.6. To stop the application, click in the integrated terminal and type `Ctrl+C`. If a prompt displays, then provide an appropriate response to terminate the process.

4.7. To clean up, click the **Kill Terminal** icon to close the integrated terminal window.

5. Commit your changes locally, and then push the new commit to your GitHub repository.

5.1. Access the Source Control view (**View** > **SCM**) and then click `DO101x-apps` in the `SOURCE CONTROL REPOSITORIES` list.

5.2. Hover over the entry for the `app.js` file under the `CHANGES` heading for the `DO101x-apps` repository. Click the `Stage Changes` icon to add the changes to the `app.js` file to your next commit.

5.3. Add a commit message of `add /mars endpoint` in the message prompt, and then click the check mark button to commit the staged changes:

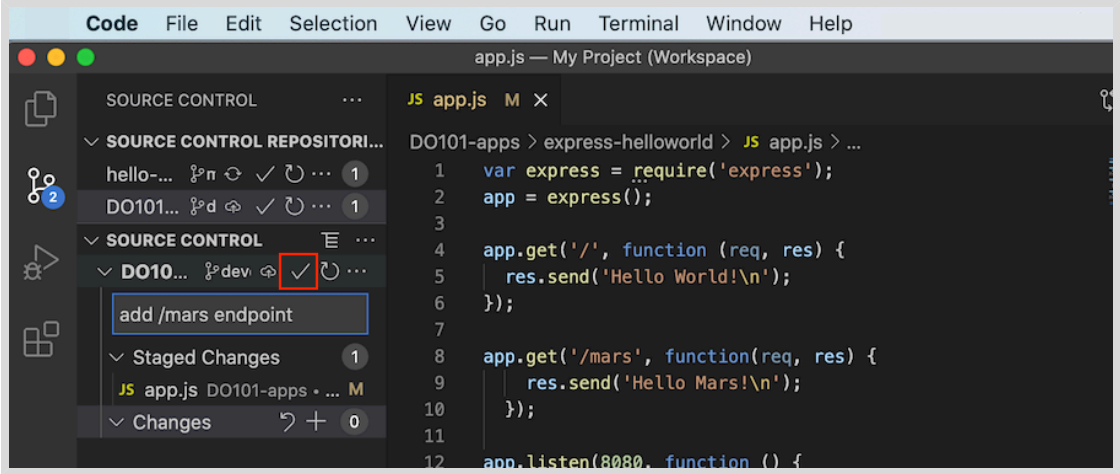


Figure 1.78: A commit message for a new feature.

6. Publish the `devenv-versioning` branch to your GitHub repository. Verify that your changes are present on GitHub.

6.1. In the Source Control view (**View** > **SCM**), locate the `DO101x-apps` entry under the `SOURCE CONTROL PROVIDERS` heading. Click the `Publish Changes` icon to publish the `devenv-versioning` branch to the remote repository. If a prompt displays, then provide your GitHub user name and password.

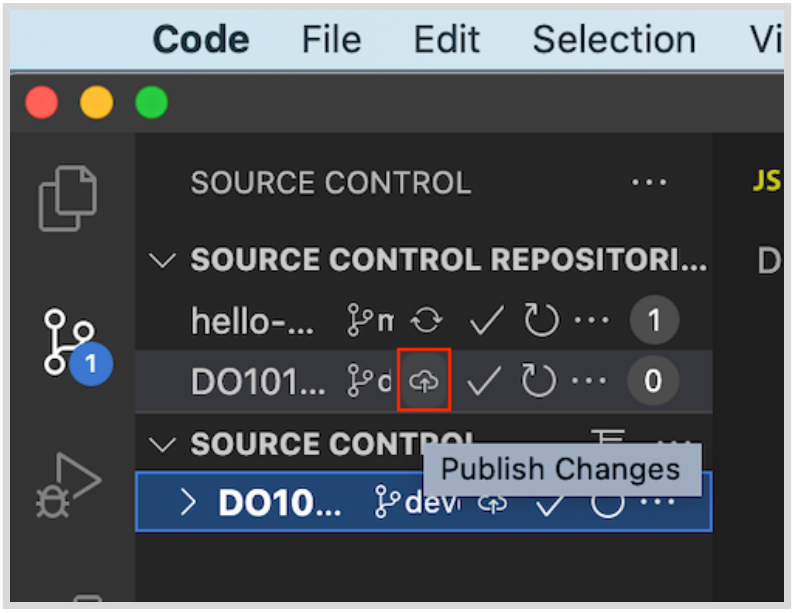


Figure 1.79: Publish a local branch to a remote repository.

6.2. In a browser, navigate to your personal `DO101x-apps` repository at `https://github.com/`*`yourgituser`*`/DO101x-apps`. The GitHub interface indicates that you recently pushed code changes to the `devenv-versioning` branch.

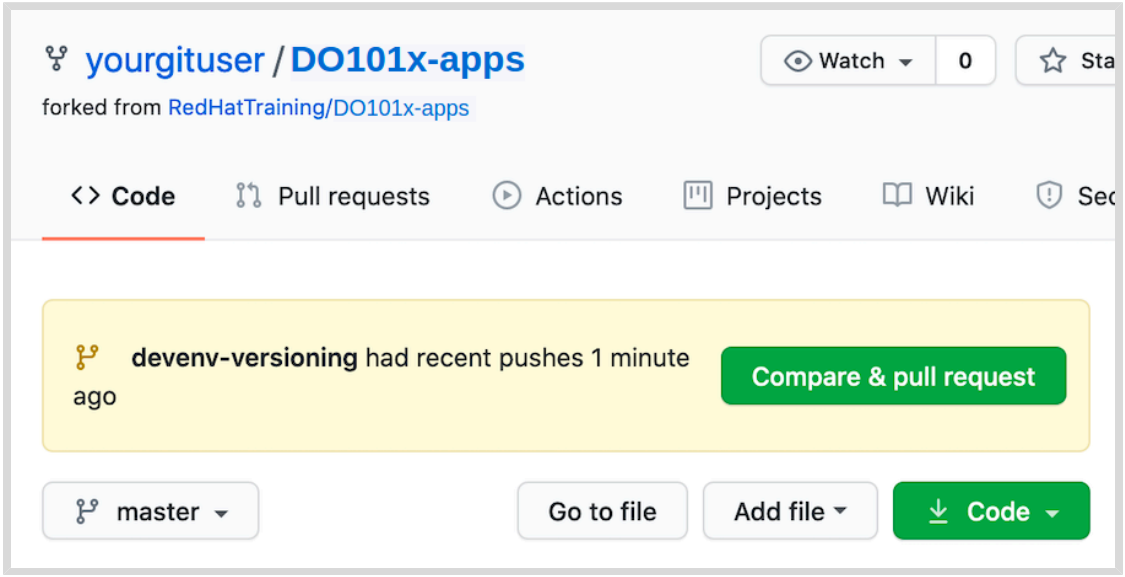


Figure 1.80: A GitHub repository indicates a recent push for the new branch.

6.3. Click `commits` below the `Code` button to display a list of recent commits.

6.4. From the branch list, select `devenv-versioning`. An entry for the `add /mars endpoint` commit displays at the top of the commit history.

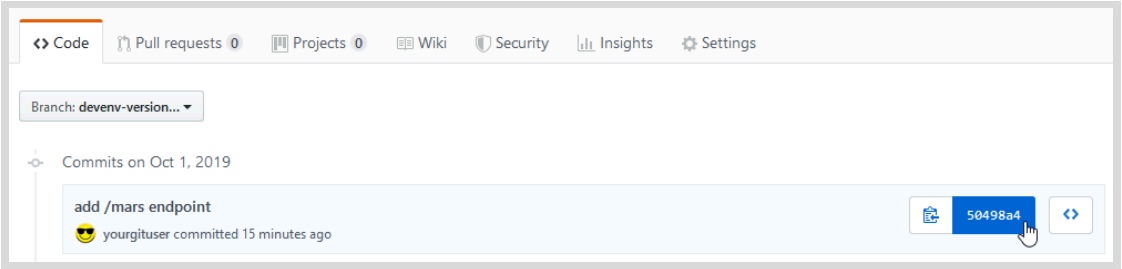


Figure 1.81: The commit history for a GitHub repository branch.

6.5. To the right of the `add /mars endpoint` entry, click the seven character commit hash. In the example shown, this value is *`50498a4`*.

GitHub displays the four lines of code that you added to the `express-helloworld/app.js` file.

Figure 1.82: GitHub displays commit code changes.

You successfully pushed your code changes in a new branch to your GitHub repository.

Finish

This concludes the guided exercise.

Summary

In this chapter, you learned:

- Integrated Development Environments (IDEs) combine separate software development tools (compilers, test suites, linters) into a single application to improve your productivity.
- Git is a common software configuration management tool used to implement version control of application code changes.
- Many source code hosting services provide access to the Git repository for a project, which enables collaboration on source code development.

Chapter 2

Deploying Applications to Red Hat OpenShift Container Platform

Goal	Deploy an application to OpenShift.
Objectives	Deploy an application to Red Hat OpenShift Container Platform.
Sections	Deploying an Application to Red Hat OpenShift Container Platform (and Guided Exercise)

Deploying an Application to Red Hat OpenShift Container Platform

Objectives

After completing this section, you should be able to deploy an application to Red Hat OpenShift Container Platform.

Introducing OpenShift Container Platform

Red Hat OpenShift Container Platform is a self-service platform where development teams can deploy their applications. The platform integrates the tools to build and run applications, and manages the complete application life cycle from initial development to production.

OpenShift offers several deployment scenarios. One typical workflow starts when a developer provides the Git repository URL for an application to OpenShift.

The platform automatically retrieves the source code from Git, and then builds and deploys the application. The developer can also configure OpenShift to detect new Git commits, and then automatically rebuild and redeploy the application.

By automating the build and deployment processes, OpenShift allows developers to focus on application design and development. By rebuilding your application with every change you commit, OpenShift gives you immediate feedback. You can detect and fix errors early in the development process, before they become an issue in production.

OpenShift provides the building mechanisms, libraries, and runtime environments for the most popular languages, such as Java, Ruby, Python, PHP, .NET, Node.js, and many more. It also comes with a collection of additional services that you can directly use for your application, such as databases.

As traffic and load to your web application increases, OpenShift can rapidly provision and deploy new instances of the application components. For the Operations team, it provides additional tools for logging and monitoring.

OpenShift Container Platform Architecture

Red Hat OpenShift Online, at `https://www.openshift.com/`, is a public OpenShift instance run by Red Hat. With that cloud platform, customers can directly deploy their applications online, without needing to install, manage, and update their own instance of the platform.

The Developer Sandbox for Red Hat OpenShift, at https://developers.redhat.com/developer-sandbox, provides an environment for developers to use OpenShift and relevant tools without needing to configure a cluster themselves. The sandbox provides a private OpenShift environment in a shared, multi-tenant OpenShift cluster that is pre-configured with a set of developer tools.

Red Hat also provides the Red Hat OpenShift Container Platform that companies can deploy on their own infrastructure. By deploying your own instance of OpenShift Container Platform, you can fine tune the cluster performance specific to your needs. In this classroom, you will be provided access to a private OpenShift cluster.

Application Architecture

Several development teams or customers can share the same OpenShift platform. For security and isolation between projects and customers, OpenShift builds and runs applications in isolated `containers`.

A container is a way to package an application with all its dependencies, such as runtime environments and libraries.

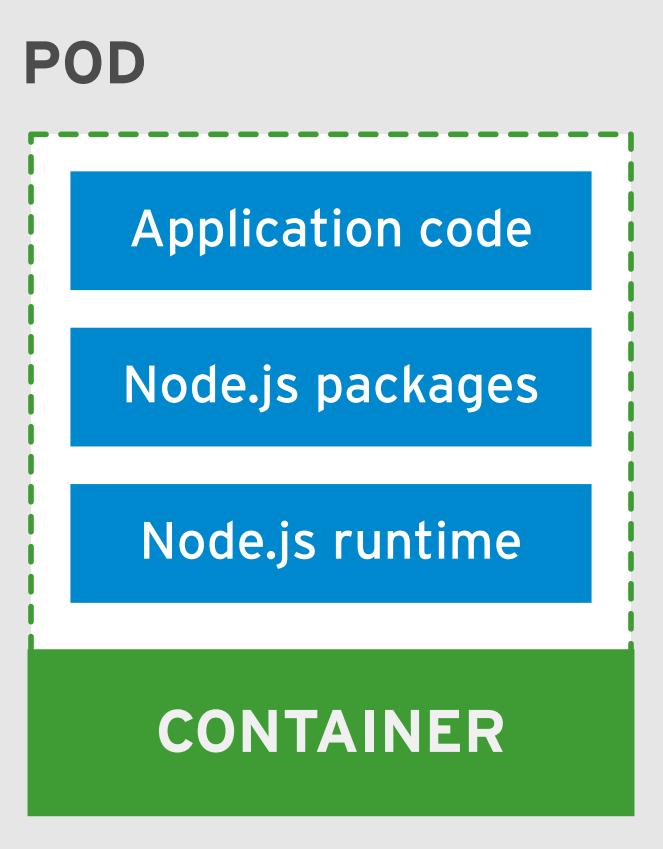


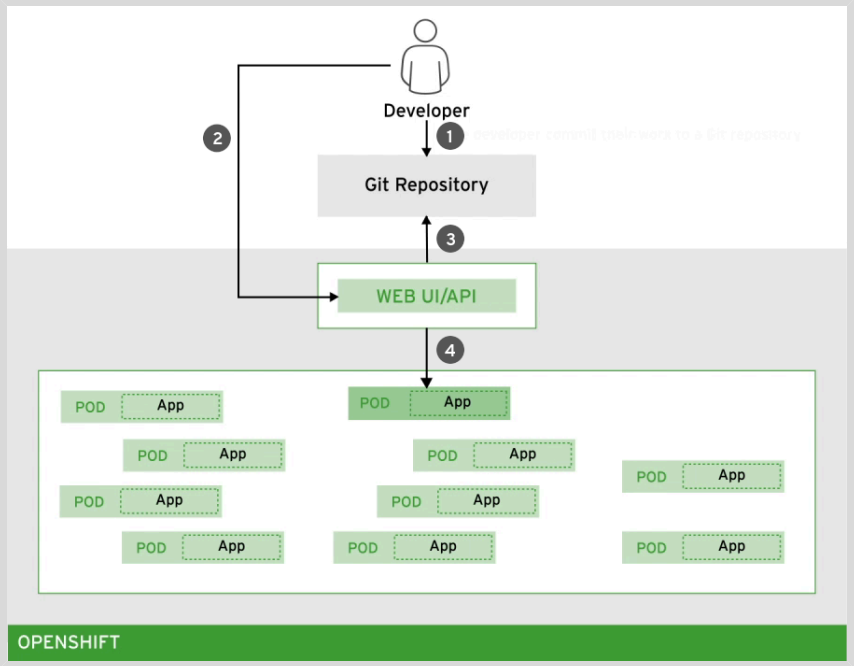
Figure 2.1: A container in a pod

The previous diagram shows a container for a Node.js application. The container groups the Node.js runtime, the Node.js packages required by the application, and the application code itself.

To manage the containerized applications, OpenShift adds a layer of abstraction known as the **pod**.

Pods are the basic unit of work for OpenShift. A pod encapsulates a container, and other parameters, such as a unique IP address or storage. A pod can also group several related containers that share resources.

The following diagram shows an OpenShift platform, hosting several applications running in pods. To deploy a new application, use the following workflow:



1. The developers commit work to a Git repository.
2. When ready to deploy their code, the developers use the OpenShift `web console` to create the application. The URL to the Git repository is one of the required parameters.
3. OpenShift retrieves the code from the Git repository and builds the application.
4. OpenShift deploys the application in a pod.

OpenShift Resource Types

OpenShift uses `resources` to describe the components of an application. When you deploy a new application, OpenShift creates those resources for you, and you can view and edit them through the web console.

For example, the Pod resource type represents a container running on the platform. A Route resource associates a public URL to the application, so your customers can reach it from outside OpenShift.

Introducing the Developer Web Console for OpenShift

The OpenShift web console is a browser-based user interface that provides a graphical alternative to the command-line tools. With the web UI, developers can easily deploy and manage their applications.

Logging in and Accessing the Developer Perspective

To access the web console, use the URL of your OpenShift platform. To use OpenShift, each developer must have an account. The following screen capture shows the login page.

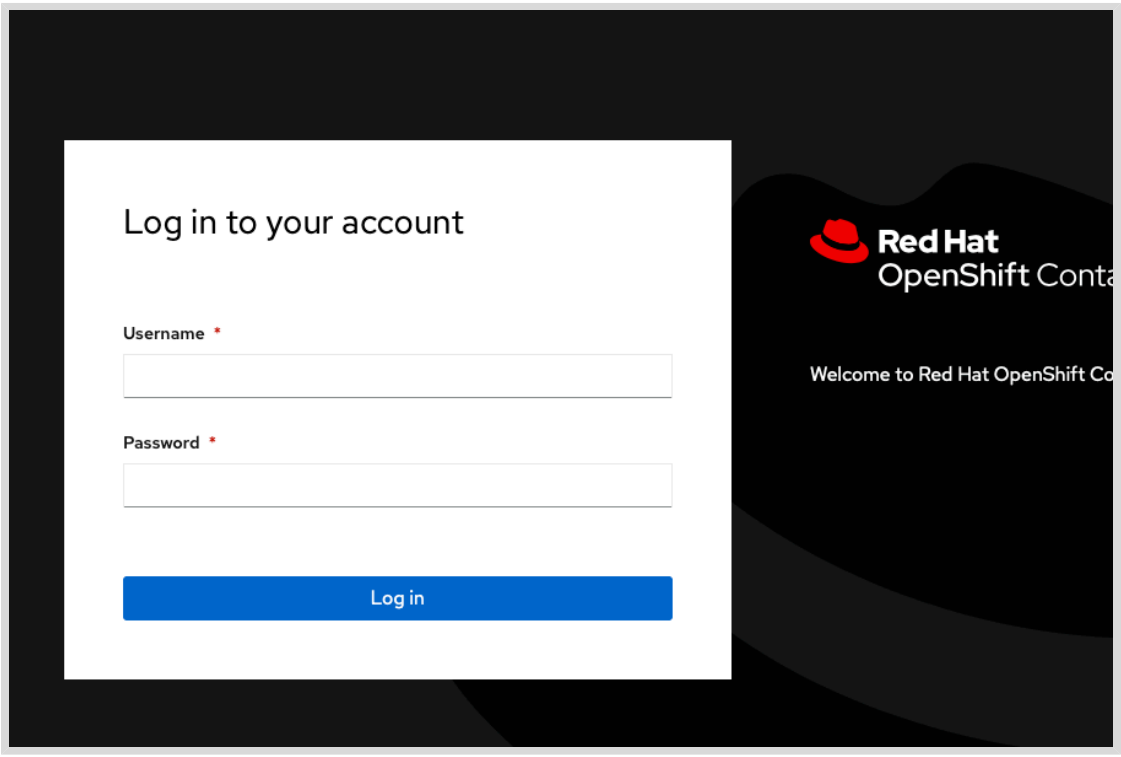


Figure 2.2: Logging in to OpenShift

The web console provides two perspectives, one for developers, and the other for administrators and operators. As shown in the following screen capture, the perspective is selected from the menu on the left. As a developer, you usually select the `Developer` perspective.

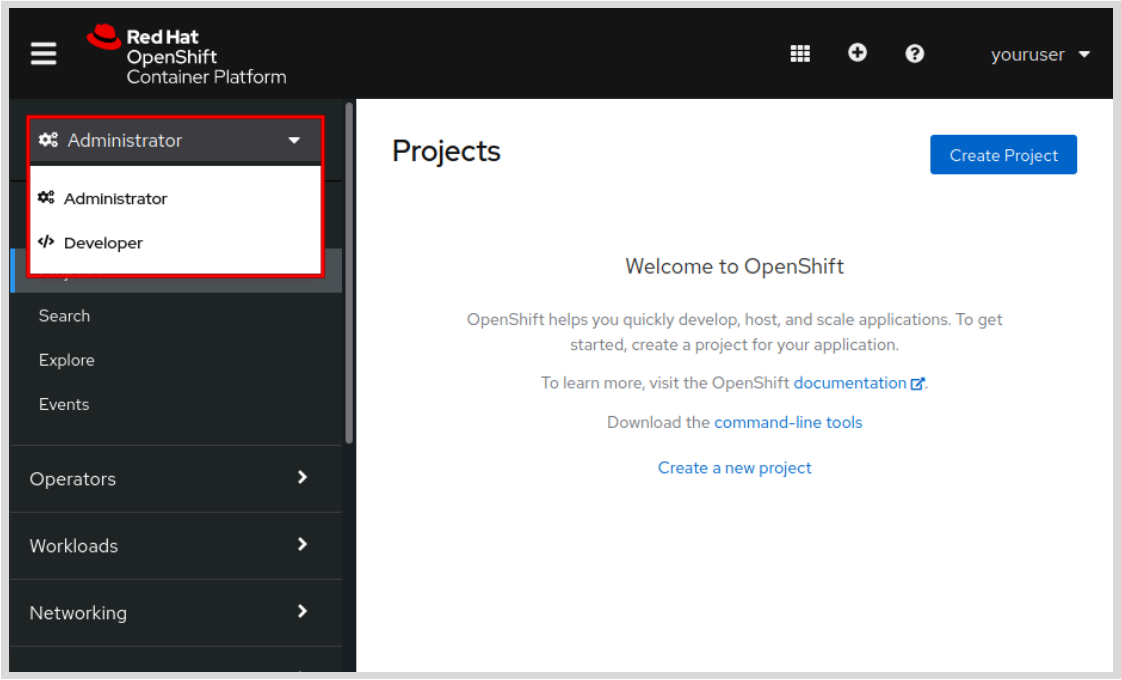


Figure 2.3: The administrator and developer perspectives

Creating a Project

OpenShift groups applications in `projects`. Using projects, developer teams can organize content in isolation from other teams. Projects enable both grouping the individual components of an application (front end, back-end, and database), and creating life cycle environments (development, QA, production).

To create a project, select **Create Project** from the `Project` list.

> **Note**
> If you do not see the `Project` list, switch to the `Developer` perspective and select `Topology` from the menu on the left.

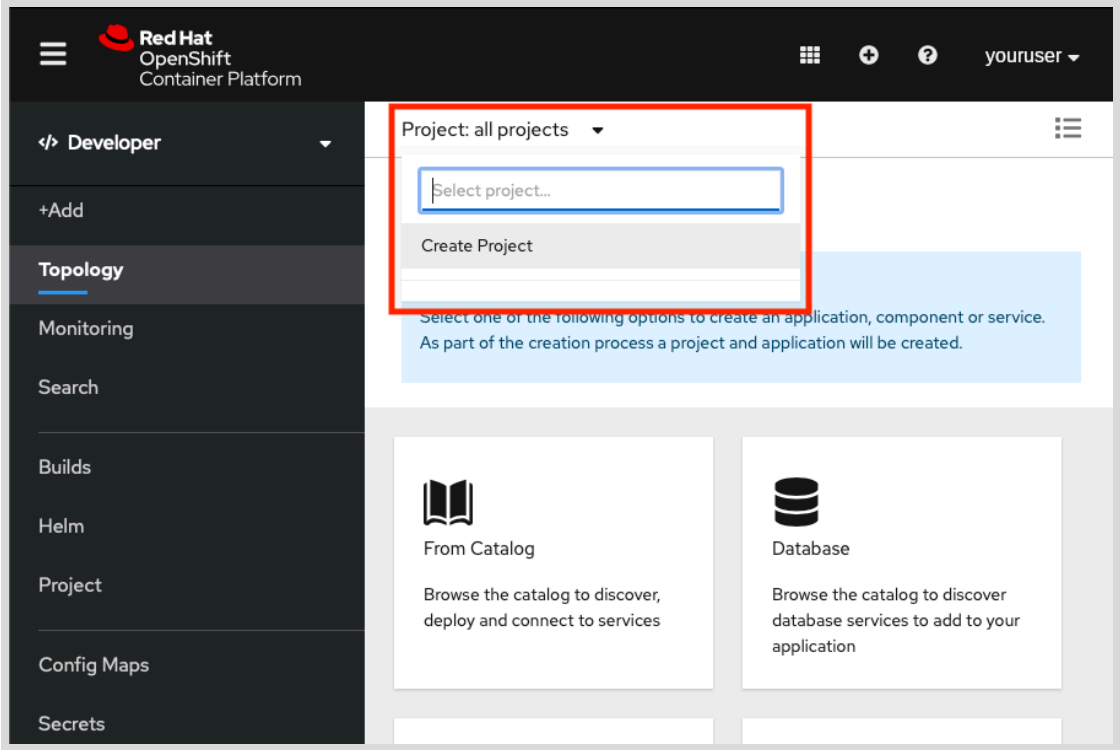


Figure 2.4: Creating a project

> **Note**
> If you use the Red Hat OpenShift Developer Sandbox, you cannot create new projects.

Deploying a New Application

OpenShift provides several methods to add a new application. The `Add` option is the entry point to an assistant that allows you to choose between the available methods to deploy an application to the OpenShift cluster as part of a specific project.

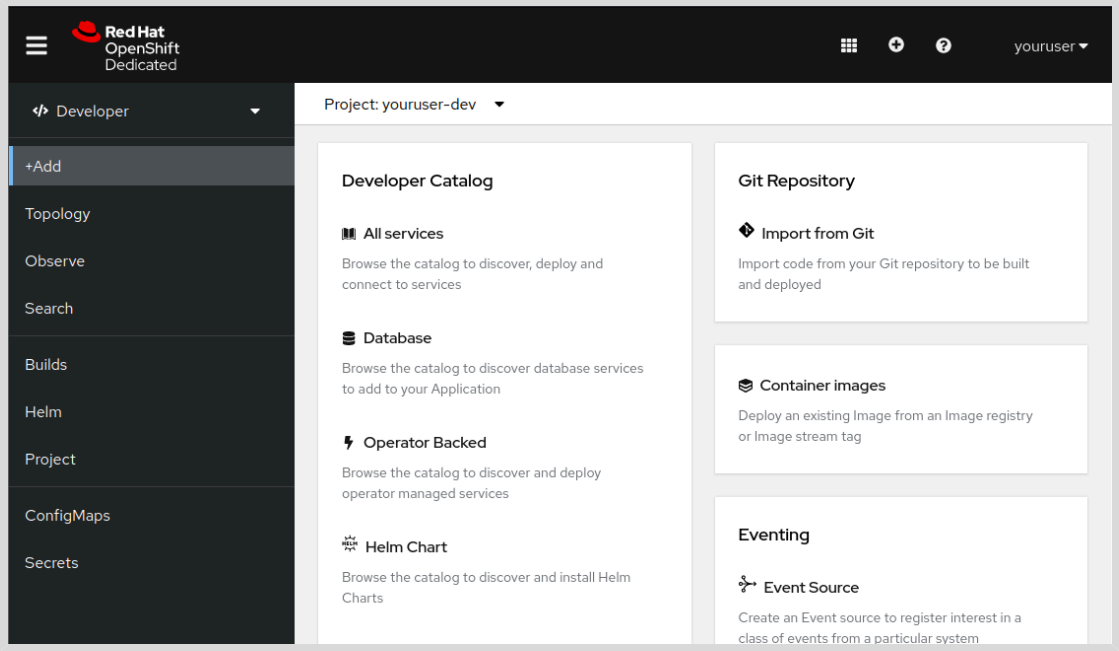


Figure 2.5: Selecting a method to deploy a new application in OpenShift

With the `Developer Catalog` method, you can list and deploy the available ready to use applications, for example, a MariaDB database. You can also select the language of your application, and provide its source code from a Git repository.

The following screen capture shows some of the available languages.

Note

You may need to select the `Template` check box under the `Type` filter to view the language templates.

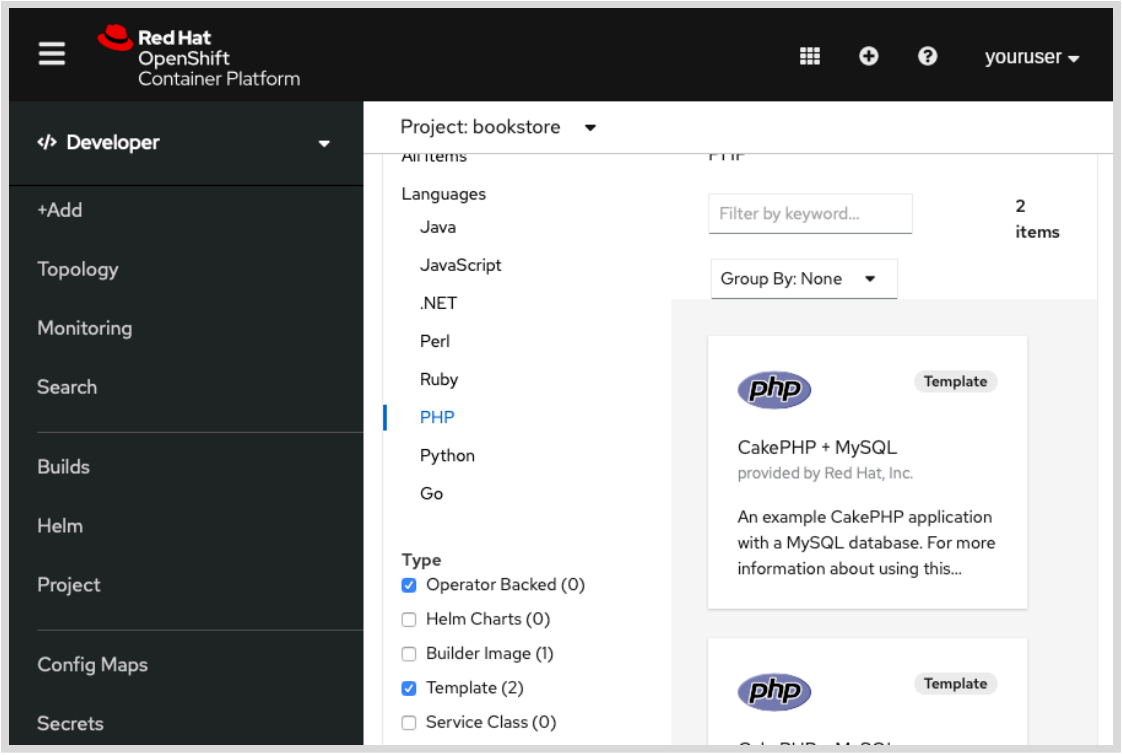


Figure 2.6: Available languages in the catalog

After selecting the application language from the catalog and clicking **Instantiate Template**, OpenShift provides a form to collect the application details.

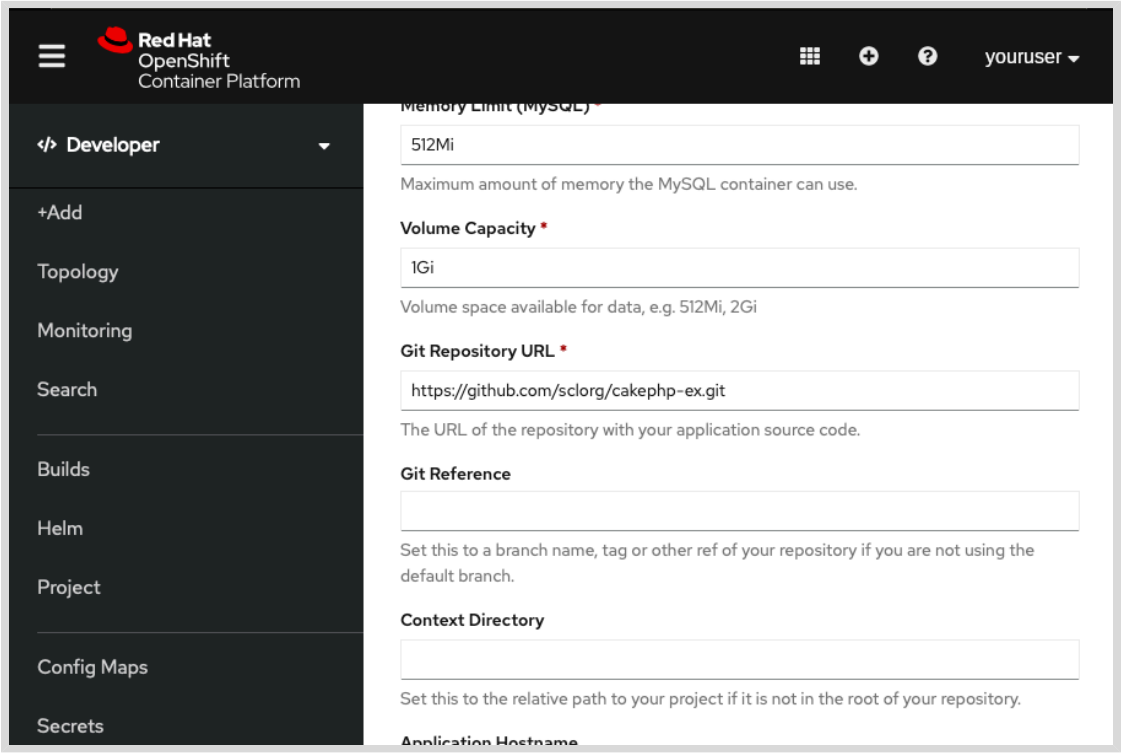


Figure 2.7: Deploying an application – Git details

Use the fields in the first section of the form to provide Git repository details for the application.

If the source code to deploy is not in the Git `main` branch, then provide the branch name in the `Git Reference` field. Use the `Context Dir` field when the application to deploy is in a subdirectory, and not at the root of the Git repository.

The `Name` field is an internal name that OpenShift uses to identify all the resources it creates during build and deployment. That name is used, for example, for the route resource that associates a public URL to the application.

Reviewing an Application in the Web Console

The `Topology` option provides an overview of the applications in a project. The following screen capture shows two applications running in the `bookstore` project: the `frontend` PHP application and a database.

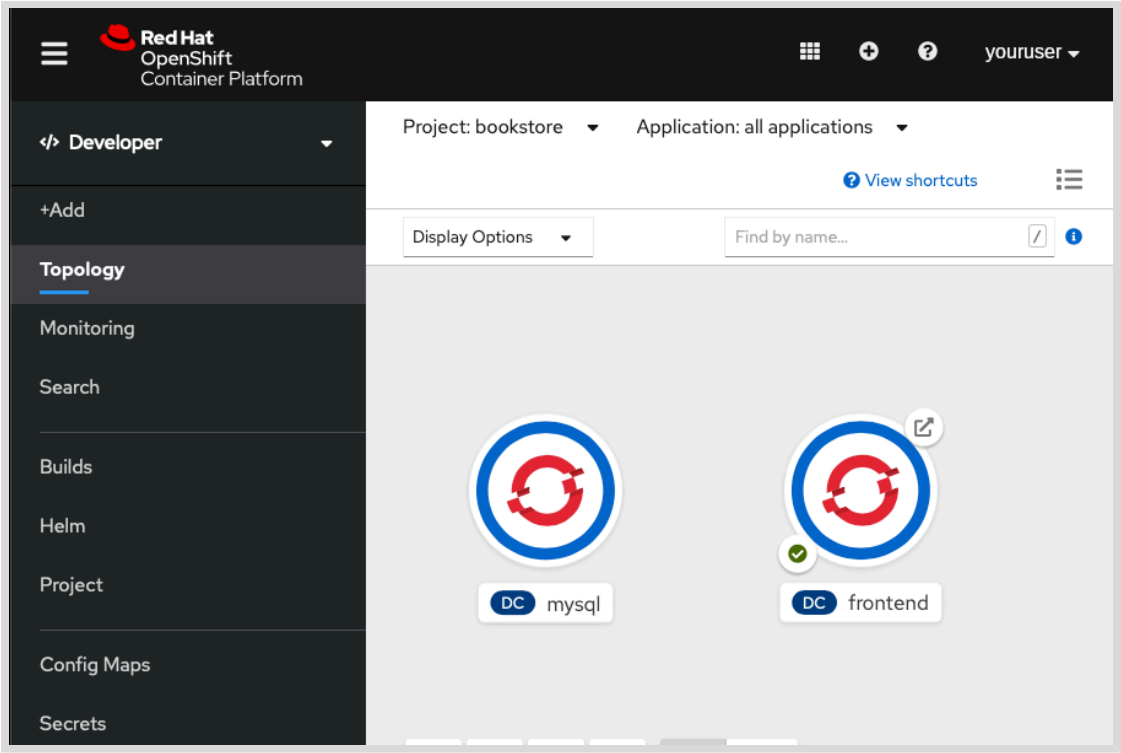


Figure 2.8: Overview of the applications in a project

Click the **application** icon to access application details. The following screen capture shows the resources that OpenShift creates for the application. Notice that one pod is running, the build is complete, and a route provides external access to the application.

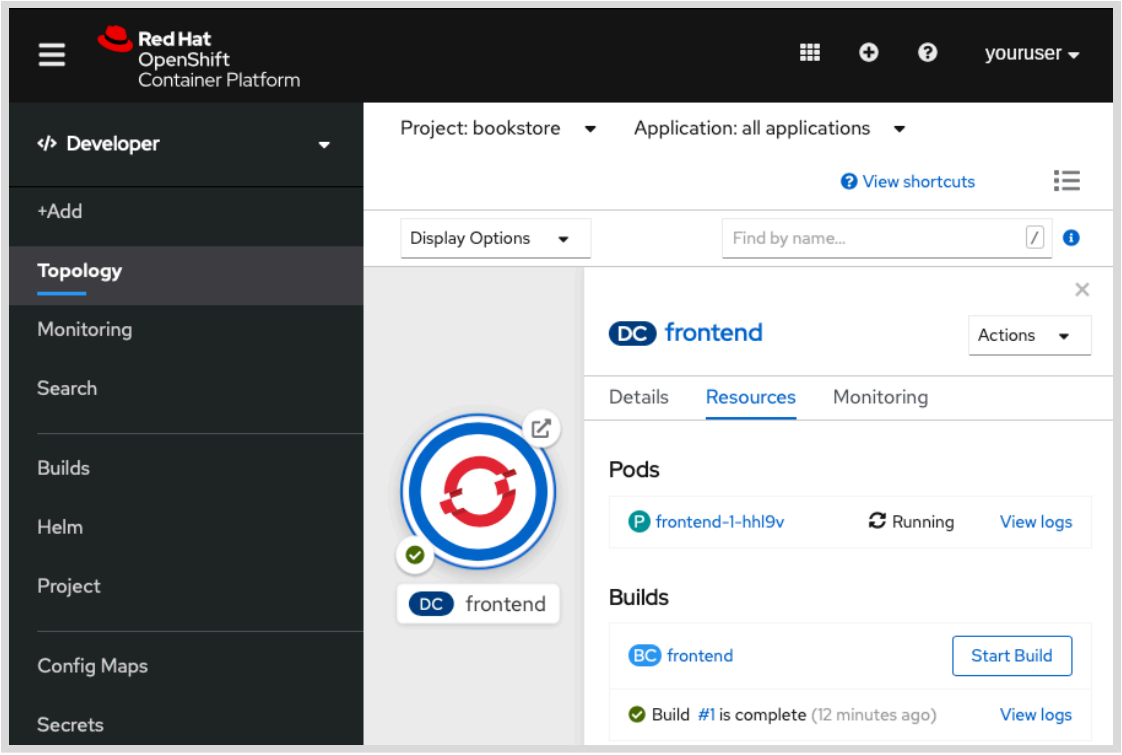


Figure 2.9: Resources of an application

Usually, when the web console displays a resource name, it is a link to the details page for that resource. The following screen capture displays the details of the Route resource.

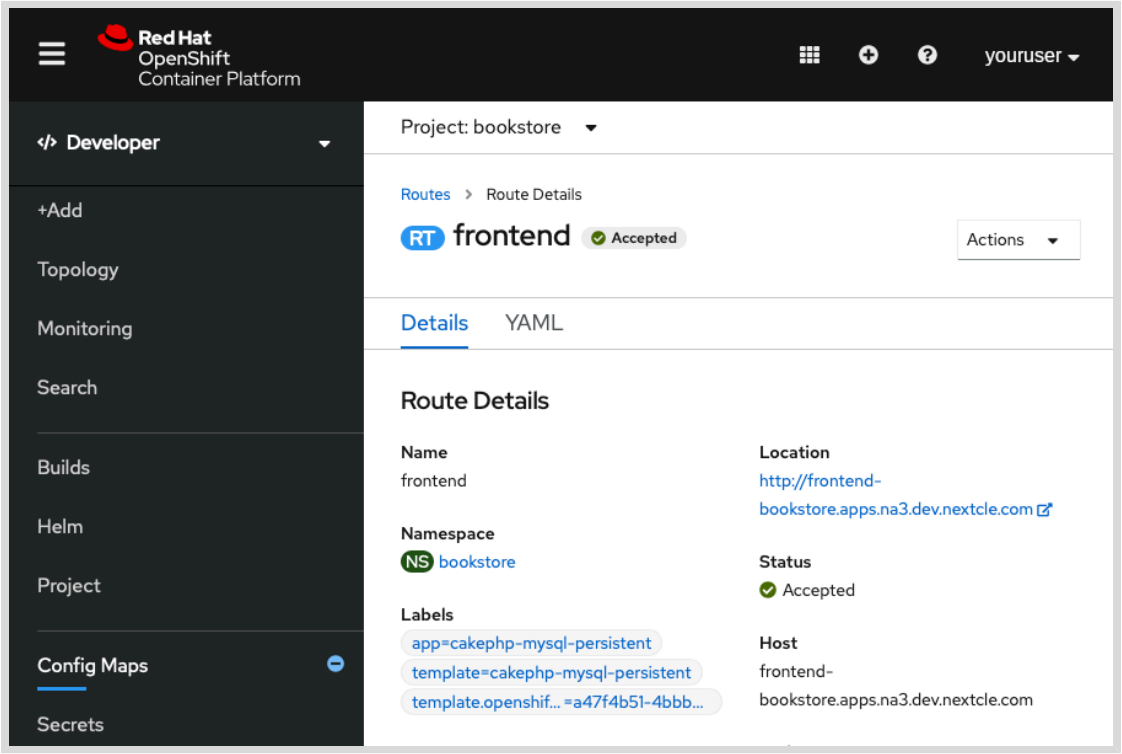


Figure 2.10: Application's route details

Editing OpenShift Resources

Most resource details pages from the OpenShift web console provide an **Actions** button that displays a menu.

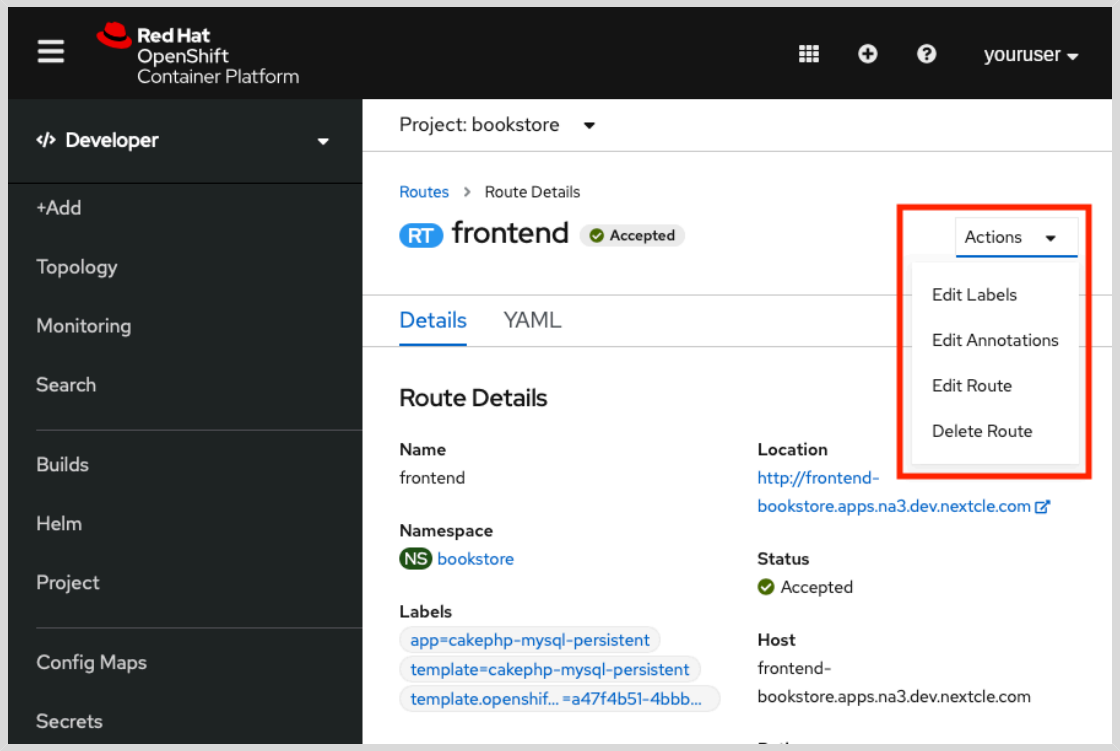


Figure 2.11: Available actions for a route resource

This menu provides options to edit and delete the resource.

References

What is OpenShift
https://cloud.redhat.com/learn/what-is-openshift

For more information, refer to the Product Documentation for Red Hat OpenShift Container Platform at
https://access.redhat.com/documentation/en-us/openshift_container_platform

Guided Exercise

Deploying an Application to Red Hat OpenShift Container Platform

In this exercise, you will use the OpenShift web console to deploy a Node.js application.

Outcomes

You should be able to use the OpenShift web console to:

- Add a new application from a Git repository.
- Inspect the resources that OpenShift creates during build and deployment.

Before You Begin

To perform this exercise, ensure that:

- The `express-helloworld` Node.js application is in your GitHub `DO101x-app` repository from the previous activity. Your changes should be in the `devenv-versioning` branch.

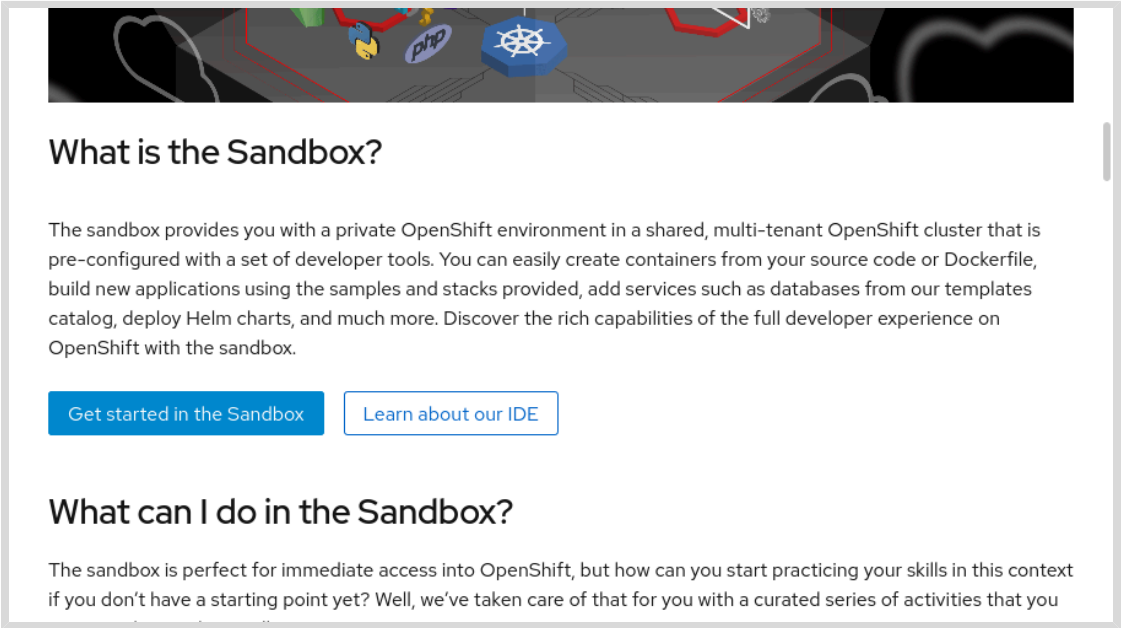
This course uses the developer sandbox for Red Hat OpenShift Container Platform (RHOCP) as a remote RHOCP cluster. Developer Sandbox for Red Hat OpenShift is a free Kubernetes-as-a-Service platform offered by Red Hat Developers, based on Red Hat OpenShift. Developer Sandbox enables users access to a pre-created RHOCP cluster.

Access is restricted to two namespaces. Developer Sandbox deletes pods after eight consecutive hours of running, and limits resources to 7 GB of RAM and 15 GB of persistent storage.

Instructions

1. Create a Developer Sandbox account.

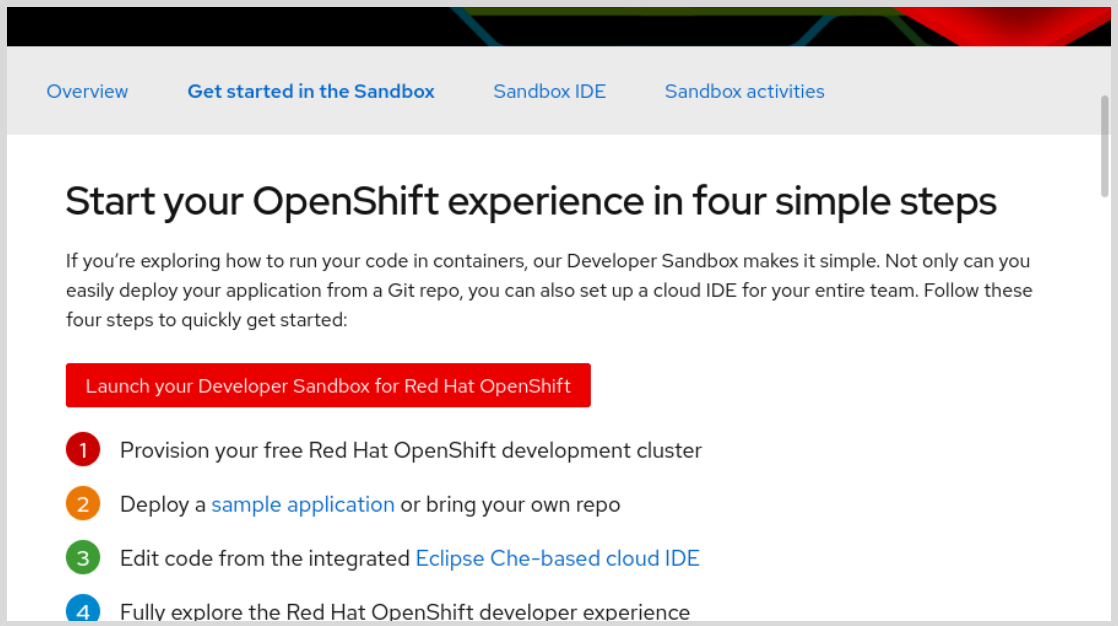
   Navigate to https://developers.redhat.com/developer-sandbox and click **Get started in the Sandbox**.



You need a free Red Hat account to use Developer Sandbox. Log in to your Red Hat account, or if you do not have one, then click **Create one now**. Fill in the form choosing a

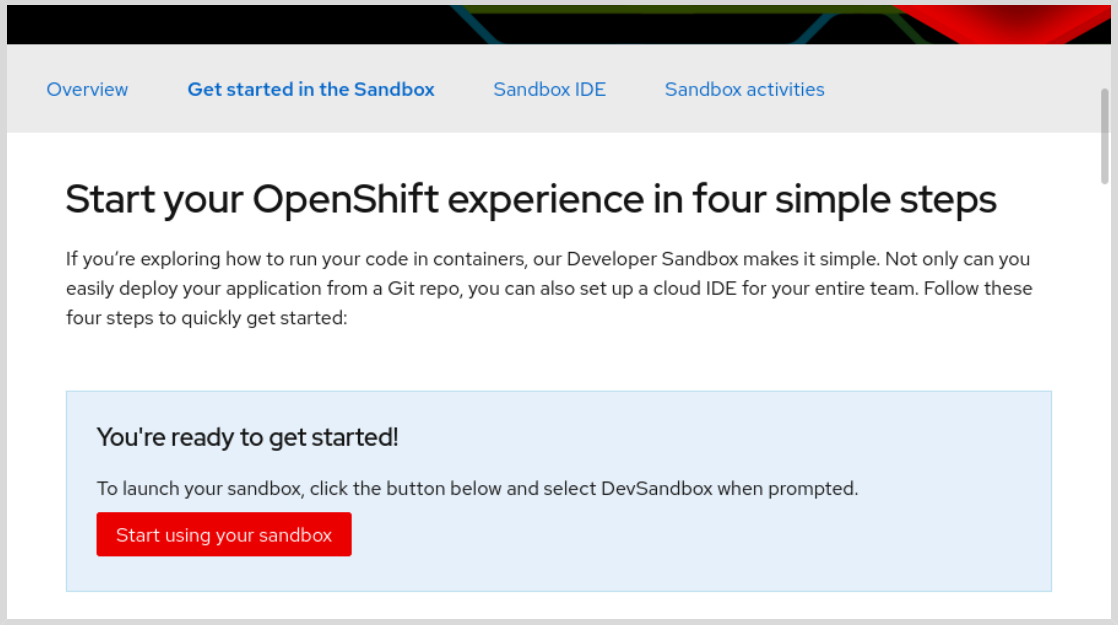
`Personal` account type, and then click **CREATE MY ACCOUNT**. You might need to accept Red Hat terms and conditions to use the Developer Program services.

When the account is ready you will be redirected back to the Developer Sandbox page. Click **Launch your Developer Sandbox for Red Hat OpenShift** to log in to Developer Sandbox.

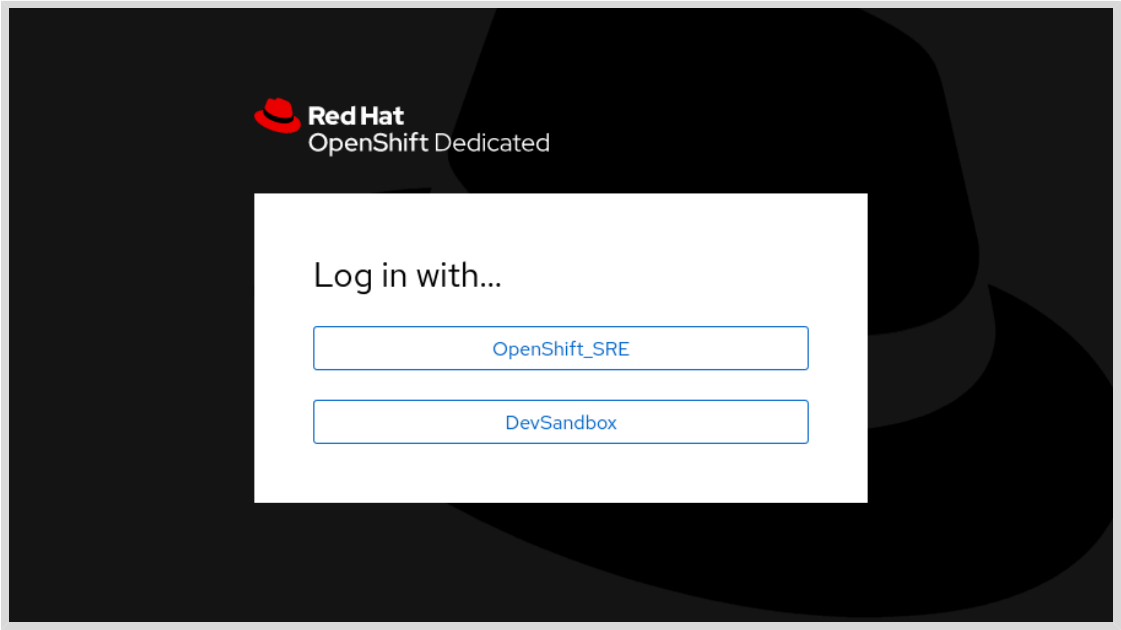


If you just created your account, then you might need to wait some seconds for account approval. You might need to verify your account via 2-factor authentication.

Once the account is approved and verified, Click **Start using your sandbox**. You might need to accept Red Hat terms and conditions to use the Developer Sandbox.

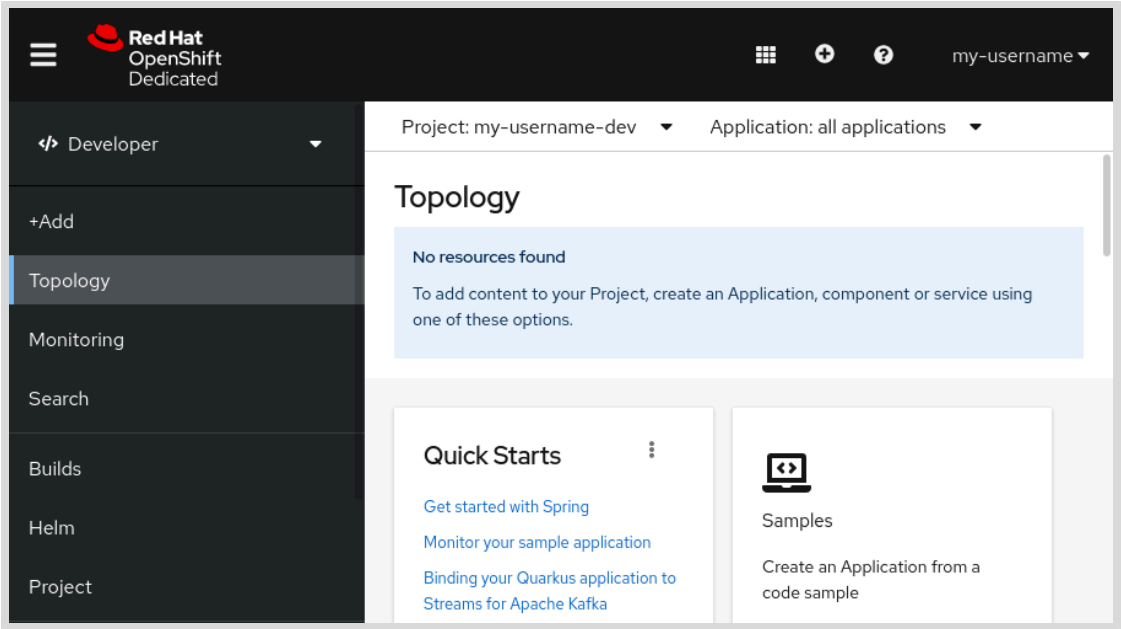


In the OpenShift log in form, click **DevSandbox** to select the authentication method.



If requested, then log in using your Red Hat account credentials.

The Red Hat OpenShift cluster and two namespaces are created for you. Only the *username*`-dev` and *username*`-stage` namespaces are available, and you can not create more.



2. Create a new JavaScript Node.js application named `helloworld`. Use the code you pushed to the Git repository in the previous exercise. The code is in the `express-helloworld` subdirectory. The branch name is `devenv-versioning`.

   2.1. Click the **Add** tab on the left side of the page and then click **All Services** in the **Developer Catalog** section.

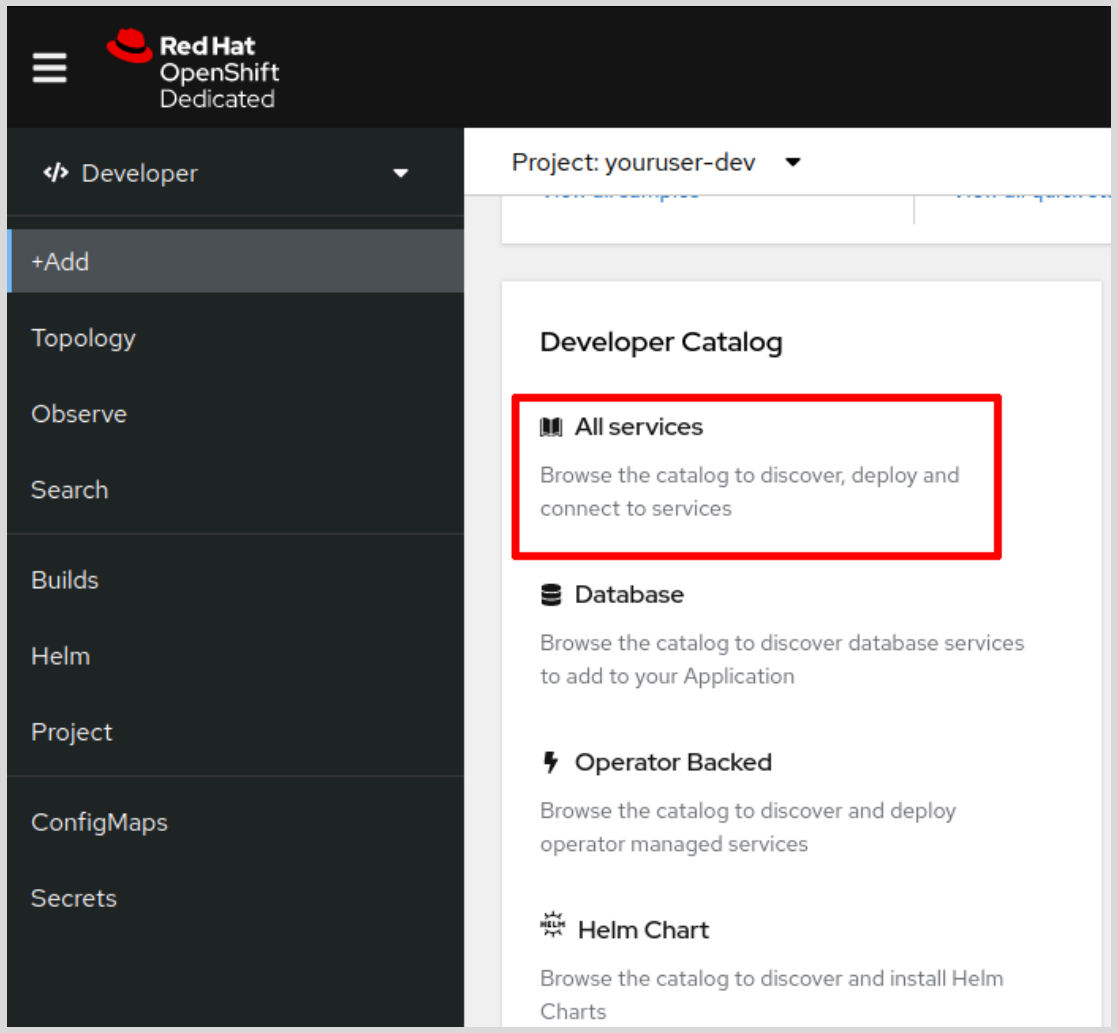


Figure 2.17: Adding a new application

2.2. Click **Languages** > **JavaScript** and then click the `Builder Image` option named `Node.js`. Click **Create Application** to enter the details of the application.

Note
You may need to select the `Builder Image` check box under the `Type` filter to view the builder image templates.

2.3. Complete the form according to the following table. To access the Git parameters, click **Show Advanced Git Options**.

New Application Parameters

Parameter	Value
`Git Repo URL`	`https://github.com/yourgituser/DO101x-apps`
`Git Reference`	`devenv-versioning`
`Context Dir`	`/express-helloworld`
`Application Name`	`helloworld`
`Name`	`helloworld`
`Resources`	`Deployment Config`
`Create a route to the application`	Make sure the box is checked

To avoid deployment errors in the following steps, review the values you entered in the form before proceeding. Click **Create** to start the build and deployment processes.

3. Wait for OpenShift to build and deploy your application.

   3.1. The web console should automatically show the `Topology` page. If necessary, click the **Topology** tab on the left side of the page.

   3.2. Wait for the application icon to change to the build complete status.

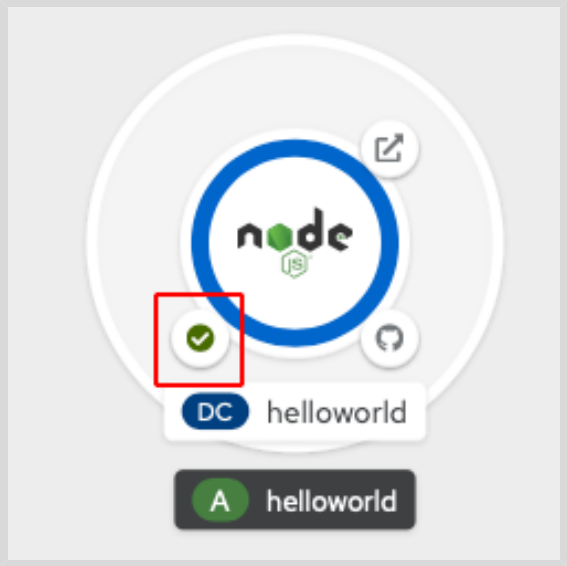


Figure 2.18: Application deployment complete

The build and deploy processes can take up to two minutes to complete.

4. Review the application resources and confirm that you can access the application from the internet.

   4.1. Click the `Node.js` icon to access the application details.

4.2. Click the **Resources** tab to list the resources that OpenShift created during the application deployment.

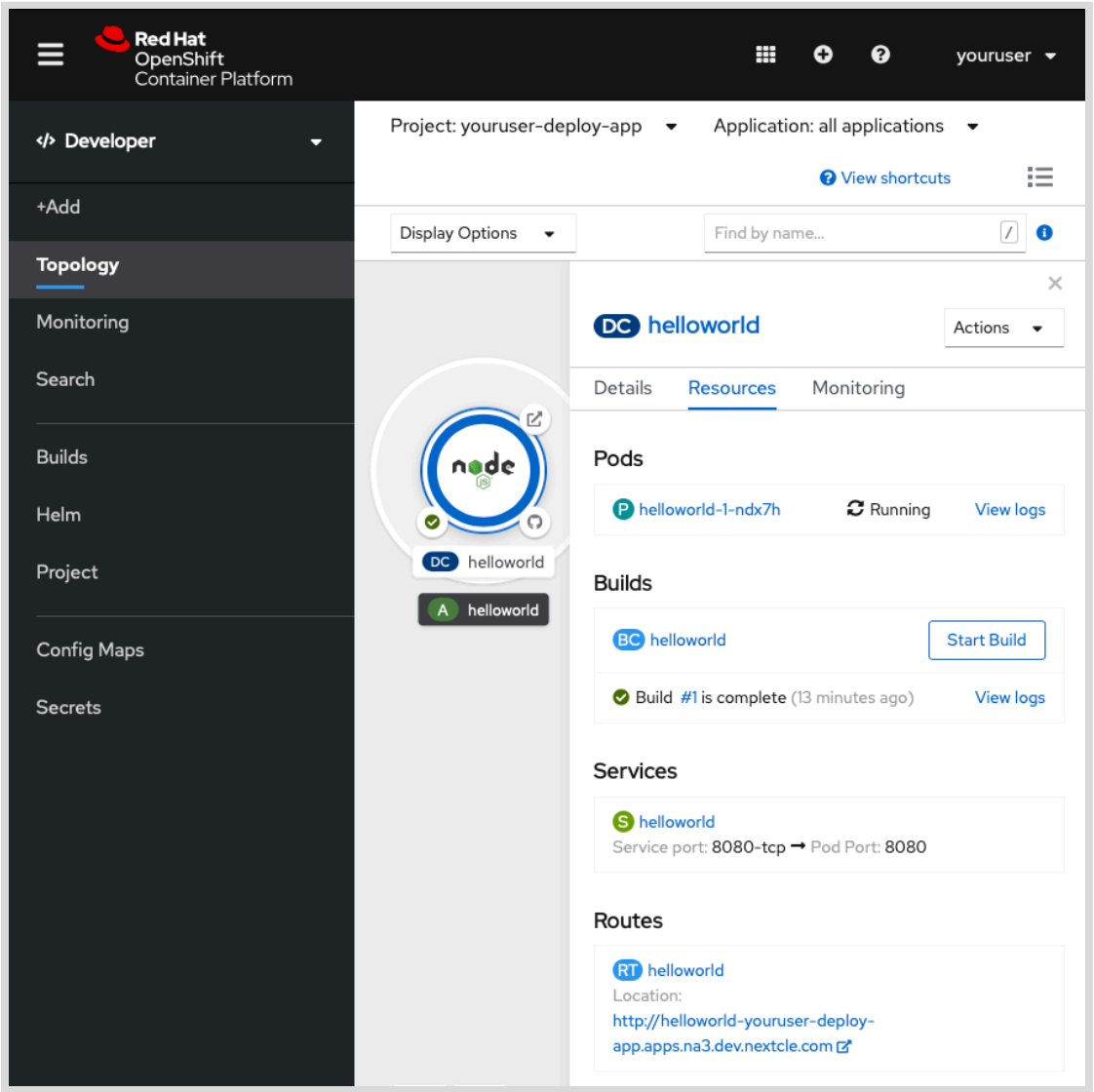


Figure 2.19: Reviewing the application resources

Notice the pod resource. This pod hosts the application running on the platform.

The build resource collects details of the application build process. Notice that the build is complete.

The route resource associates a public URL to the application.

4.3. Click the `Location` link in the route resource. Your web browser opens a new tab and accesses the application public URL.

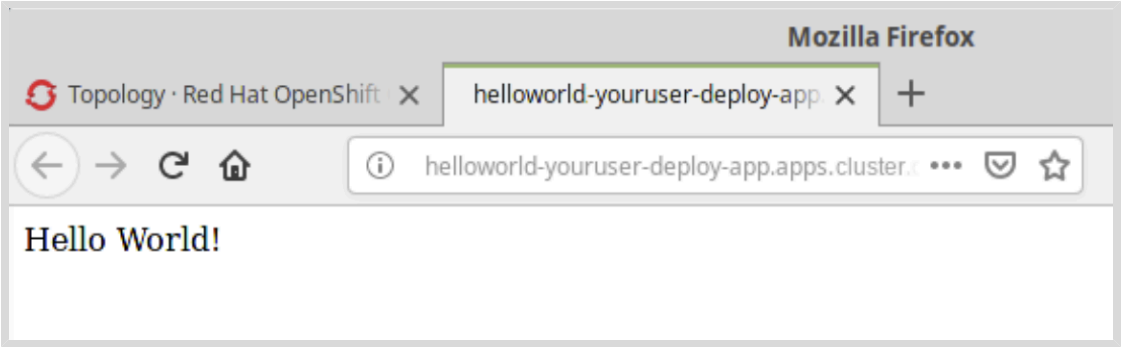


Figure 2.20: Accessing the helloworld application

When done, close the browser tab.

5. Delete the `helloworld` RHOCP objects.

   5.1. Click **Project** in the RHOCP web console. Then, click **Route**.

   5.2. Click the context menu and select **Delete Route**.

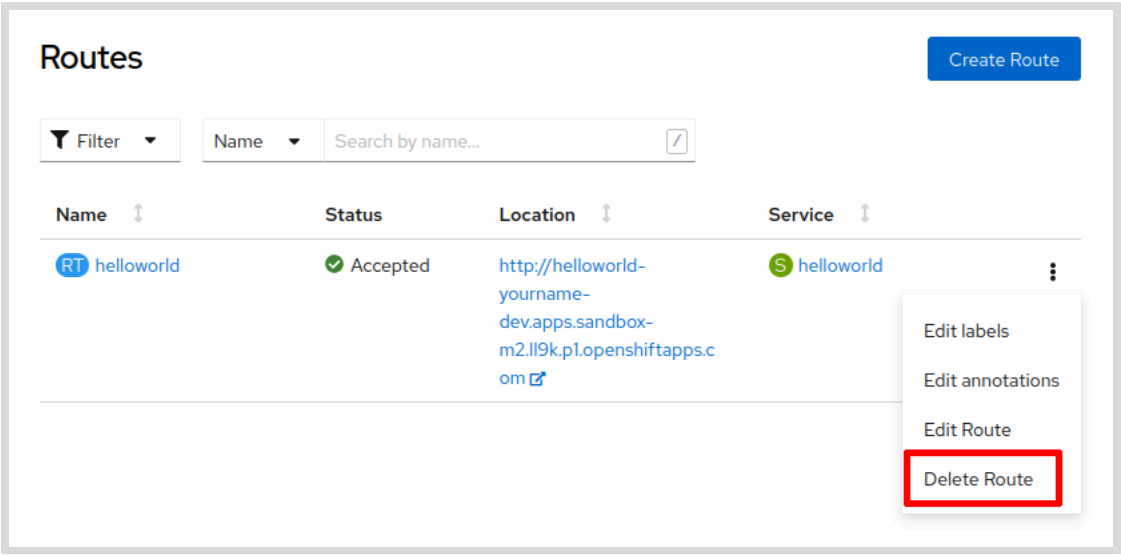


Figure 2.21: Deleting the helloworld route

   5.3. Click **Delete** in the **Delete Route?** confirmation window.

   5.4. Click **Project** in the RHOCP web console. Then, click **Service**. Repeat the previous steps to delete the `helloworld` service.

   5.5. Click **Project** in the RHOCP web console. Then, click **DeploymentConfig**. Repeat the previous steps to delete the `helloworld` deployment config object.

   Ensure the **Delete dependent objects of this resource** checkbox is selected when you confirm the deletion of the object.

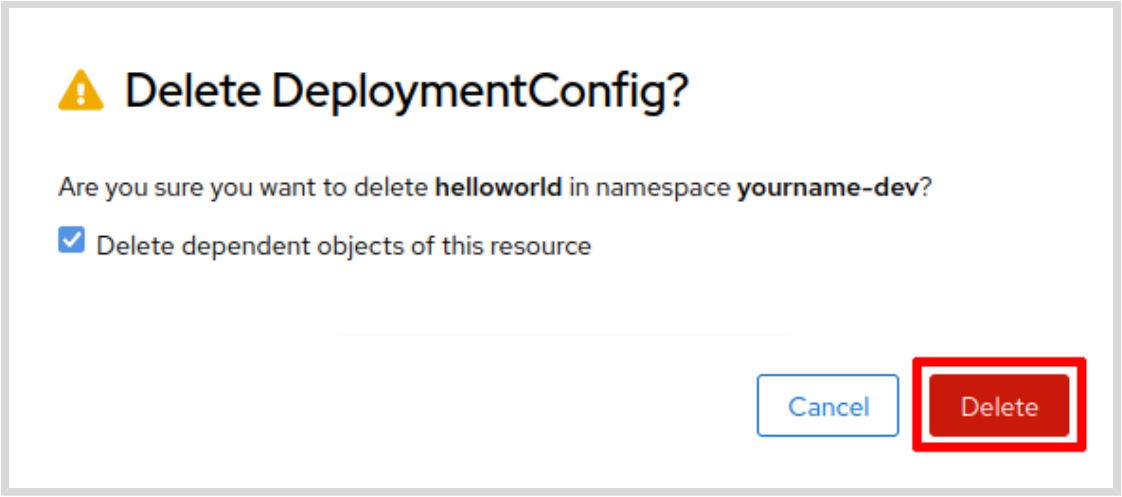


Figure 2.22: Deleting the helloworld deploymentconfig resource

5.6. Click **Builds** in the RHOCP web console. Then, repeat the previous steps to delete the `helloworld` buildconfig object.

5.7. Click **Secrets** in the RHOCP web console. Then, repeat the previous steps to delete the `helloworld-generic-webhook-secret` and `helloworld-github-webhook-secret` secrets.

Finish

This concludes the guided exercise.

Summary

In this chapter, you learned:

- OpenShift builds and runs applications in isolated pods.
- OpenShift supports building and deploying an application from source code in a Git repository.
- The web console provides a developer perspective for developers to create and manage their applications.
- A route resource associates a DNS host name to an application running in OpenShift.

Chapter 3

Configuring Application Builds in OpenShift

Goal

Manage application builds in Red Hat OpenShift Container Platform.

Objectives

- Update an application deployed on Red Hat OpenShift Container Platform.
- Add and adjust application configuration and secrets for applications deployed on Red Hat OpenShift Container Platform.
- Deploy an application that connects to a database on the Red Hat OpenShift Container Platform.

Sections

- Configuring Application Secrets (and Guided Exercise)
- Connecting an Application to a Database (and Guided Exercise)

Updating an Application

Objectives

After completing this section, you should be able to update an application deployed on Red Hat OpenShift Container Platform.

Building and Updating Applications

To deploy applications on OpenShift, you must create a **container image**. A container image is a binary package containing an application and all of its dependencies, including the operating system.

In OpenShift, a **build** is the process of creating a runnable container image from application source code. A **BuildConfig** resource defines the entire build process.

OpenShift can create container images from source code without the need for tools such as Docker or Podman. After they are built, application container images are stored and managed from a built-in **container registry** that comes bundled with the OpenShift platform.

OpenShift supports many different ways to build a container image. The most common method is called **Source to Image** (S2I). In an S2I build, application source code is combined with an **S2I builder image**, which is a container image containing the tools, libraries, and frameworks required to run the application.

For example, to run Node.js applications on OpenShift, you will use a Node.js S2I builder image. The Node.js S2I builder image is a container image configured with the Node.js runtime, package management tools (NPM), and other libraries required for running Node.js applications.

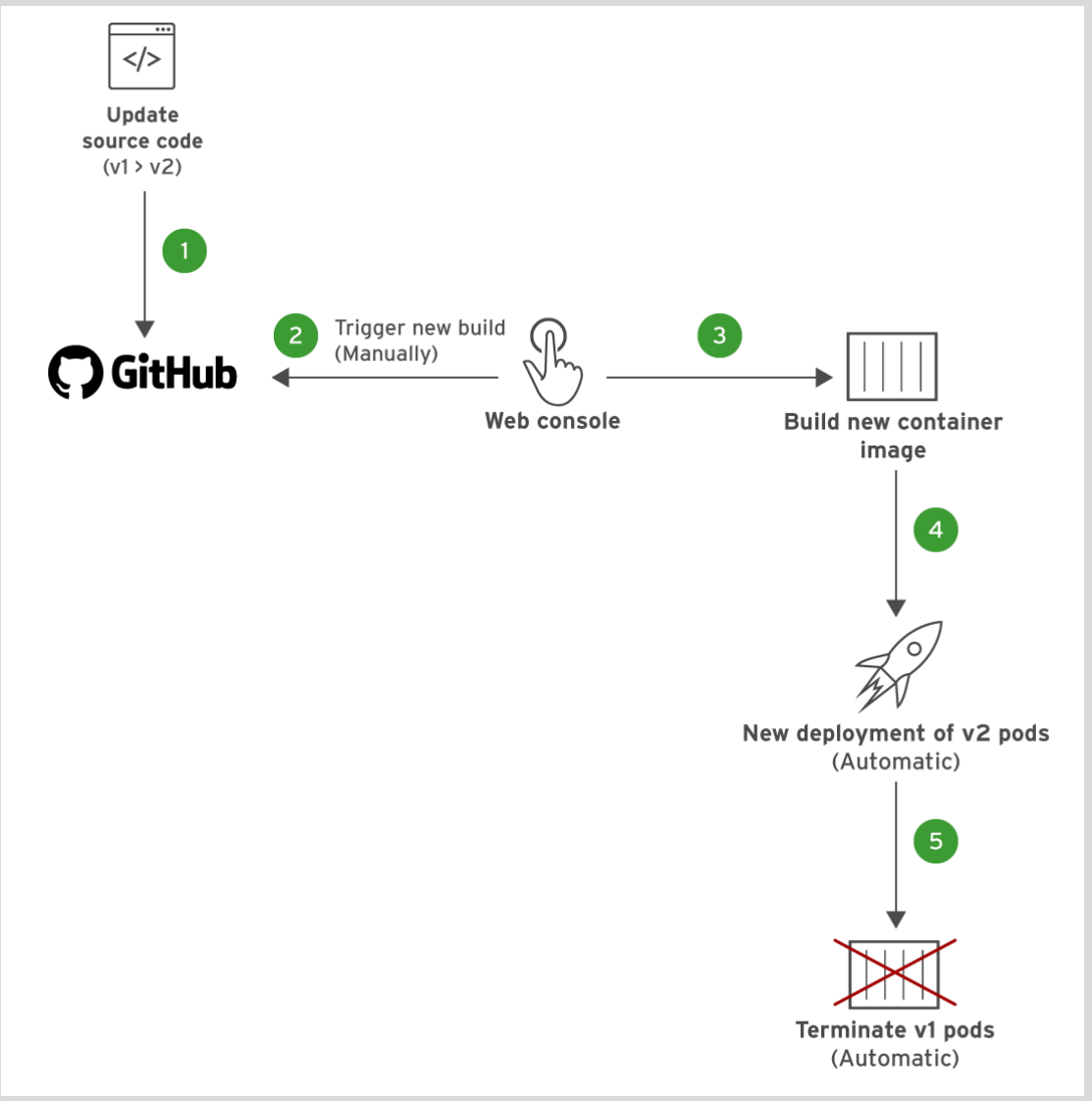
OpenShift can detect the type of application, and choose an appropriate S2I builder image to build the final application container image.

For example, if the root of the application source code tree contains a `package.json` file, OpenShift will select the latest Node.js S2I builder image for the build. You can override this default selection and choose your own S2I builder image for the build process.

Manually Triggering Builds

After an application is deployed on OpenShift, then OpenShift can rebuild and redeploy a new container image anytime the application source code is modified. Use either the `oc` command line client, or the OpenShift web console to trigger a new build of the updated application.

The workflow for an application deployed from GitHub when using the manual build process is as follows:



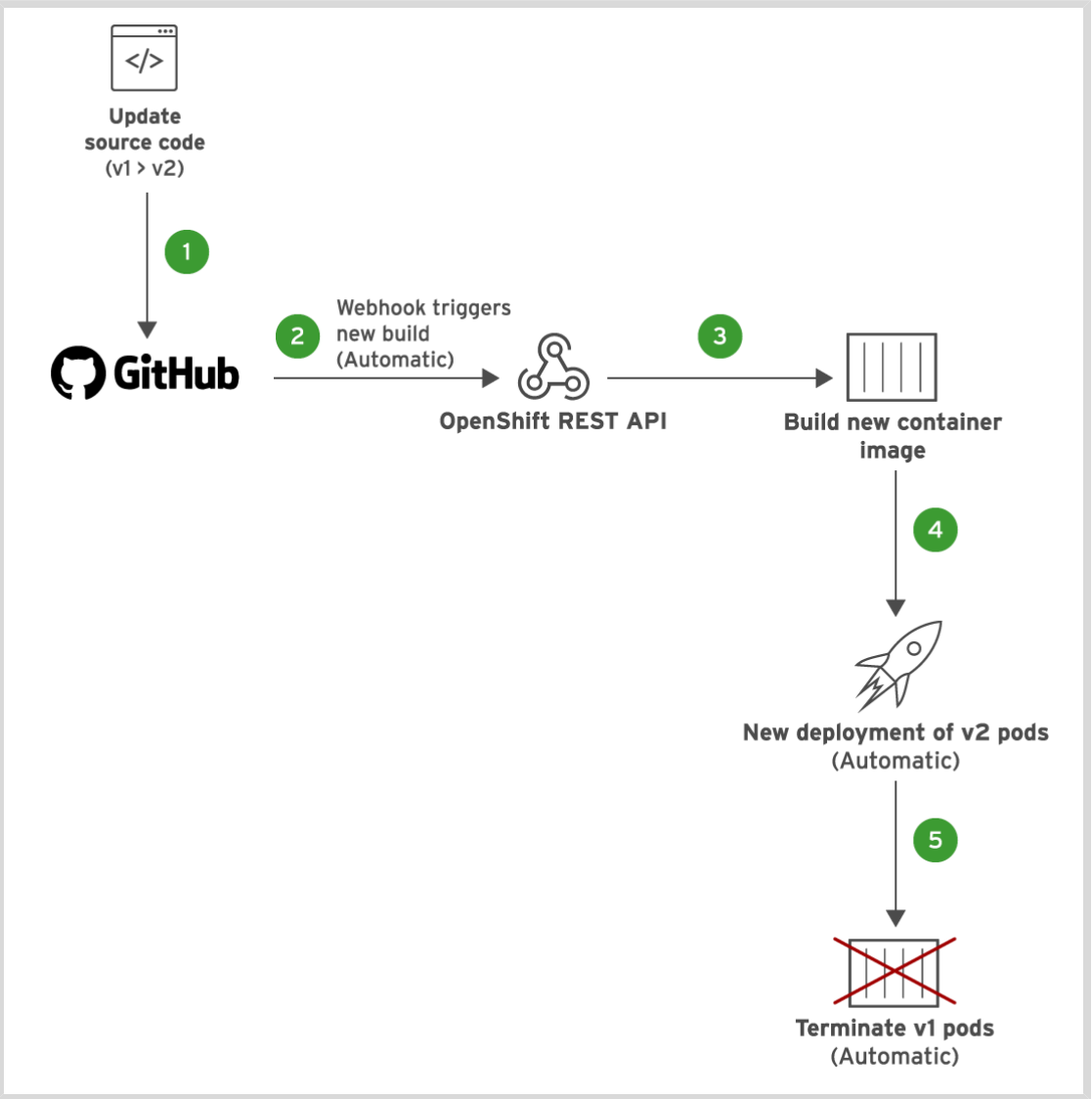
1. Update source code for an existing application, such as from v1 to v2, and then commit the changes to GitHub.
2. Manually trigger a new build using the OpenShift web console, or the OpenShift client command line interface (CLI).
3. OpenShift builds a new container image with updated code.
4. OpenShift rolls out new pods based on the new container image (v2).
5. After the new pods based on v2 are rolled out, OpenShift directs new requests to the new pods, and terminates the pods based on the older version (v1).

Automatic Builds using Webhooks

A **webhook** is a mechanism to subscribe to events from a source code management system, such as GitHub. OpenShift generates unique webhook URLs for applications that are built from source stored in Git repositories. Webhooks are configured on a Git repository.

Based on the webhook configuration, GitHub will send a HTTP POST request to the webhook URL, with details that include the latest commit information. The OpenShift REST API listens for webhook notifications at this URL, and then triggers a new build automatically. You must configure your webhook to point to this unique URL.

The workflow for an automatic rebuild triggered by webhooks is as follows:



1. Update source code for an existing application (from v1 to v2) and commit the changes to GitHub.
2. The GitHub webhook sends an event notification to the OpenShift REST API.
3. OpenShift builds a new container image with the updated code.
4. OpenShift rolls out new pods based on the new container image (v2).
5. After the new pods based on v2 are rolled out, OpenShift directs new requests to the new pods, and terminates the pods based on the older v1.

References

For more information on OpenShift builds, refer to the `Builds` chapter in the Product Documentation for Red Hat OpenShift Container Platform at
https://docs.openshift.com/container-platform/4.6/welcome/index.html

Build Triggers

https://docs.openshift.com/container-platform/4.6/builds/triggering-builds-build-hooks.html

Creating GitHub Webhooks

https://developer.github.com/webhooks/creating/

Guided Exercise

Updating an Application

In this exercise, you will update the source code for a Node.js application deployed on OpenShift.

Outcomes

You should be able to:

- Create a new Red Hat OpenShift Container Platform (RHOCP) application using the `oc new-app` command.
- Trigger a new build manually from the OpenShift web console, after updating the source code of an application.
- Set up webhooks in GitHub to automatically trigger new builds when there are new commits to the Git repository.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running Red Hat OpenShift Container Platform (RHOCP) cluster. See *Guided Exercise: Deploying an Application to Red Hat OpenShift Container Platform* for more information about using the developer sandbox RHOCP cluster.
- The source code for the `version` application in the `DO101x-apps` Git repository on your local system.

Instructions

1. Install the Red Hat OpenShift Container Platform (RHOCP) command line interface (CLI).

1.1. In the RHOCP web console, click the question mark icon next to your username in the top right. Then click **Command Line Tools**.

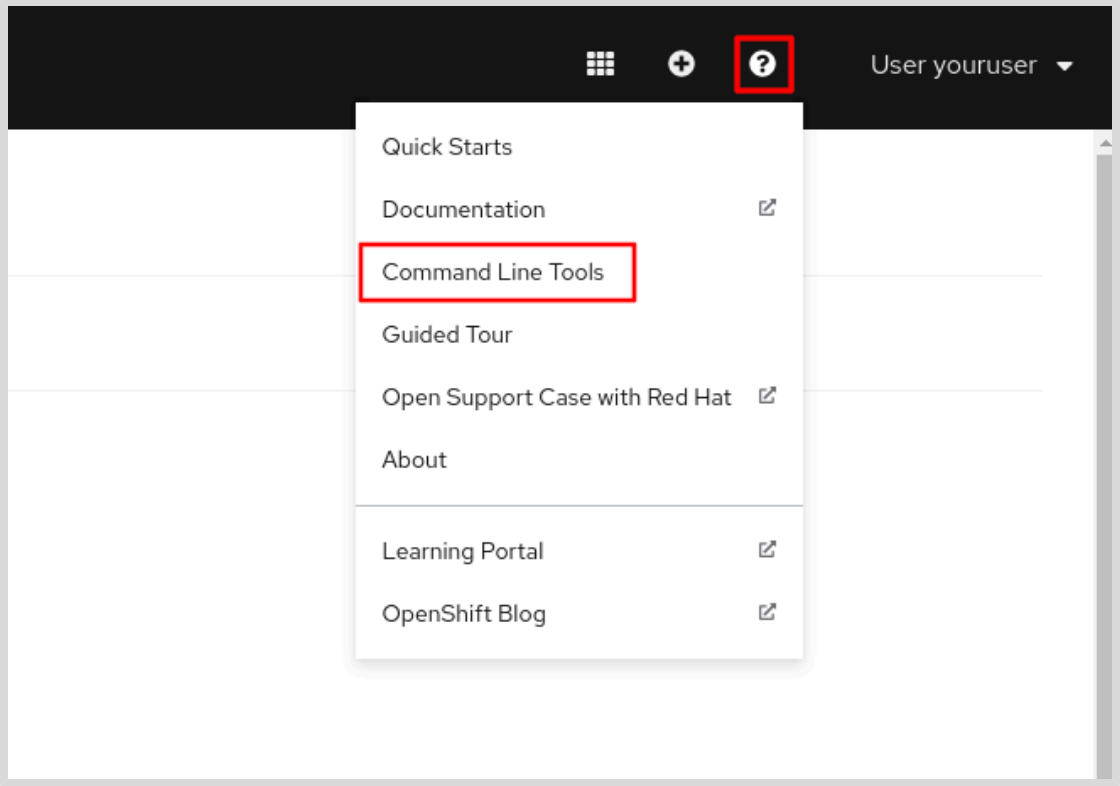


Figure 3.1: Navigate to the command line tools page

1.2. Download the relevant `oc` archive for your platform.

1.3. Unzip the compressed archive file, and then copy the `oc` binary to a directory of your choice. Ensure that this directory is in the `PATH` variable for your system.

- On macOS and Linux, copy the `oc` binary to `/usr/local/bin` and make it executable:

```
$ sudo cp oc /usr/local/bin/
$ sudo chmod +x /usr/local/bin/oc
```

- For Windows 10 systems, decompress the downloaded archive into a directory of your choice. Then, add the full path of the of the directory with the `oc` binary to your `PATH` environment variable. Follow the instructions at https://docs.microsoft.com/en-us/previous-versions/office/developer/sharepoint-2010/ee537574(v=office.14)#to-add-a-path-to-the-path-environment-variable to edit the `PATH` environment variable.

1.4. Verify that the `oc` binary works for your platform. Open a new command line terminal and run the following:

```
$ oc version --client
Client Version: 4.9.0-202110182323 ...output omitted...
```

Note

If you edit the PATH variable, restart VS Code in order to use the `oc` command from within the VS Code embedded terminal.

2. Deploy the `version` application to RHOCP by using the `oc` tool.

    2.1. In the RHOCP web console, click your username at the top right, and then click **Copy Login Command**. This will open a new tab and prompt you for your username and password.

    2.2. Once authenticated, click **Display Token** and copy the provided `oc login` command.

Important

Because this token is used for authentication, treat it as your RHOCP password.

    2.3. Log in to your RHOCP account by using the copied command and notice the default project:

```
$ oc login --token=yourtoken --server=https://api.region.prod.nextcle.com:6443
Logged into "https://api.region.prod.nextcle.com:6443" as "youruser" using the
 token provided.
...output omitted...
Using project "youruser-dev".
```

    2.4. Launch the Visual Studio Code (VS Code) editor. In the Explorer view (**View** > **Explorer**), open the `DO101x-apps` folder in the `My Projects` workspace. The source code for the `version` application is in the `version` directory.

    2.5. Ensure that the `DO101x-apps` repository uses the `main` branch. If you were working with another branch for a different exercise, click the current branch and then select `main` in the `Select a ref to checkout` window to switch to the `main` branch.

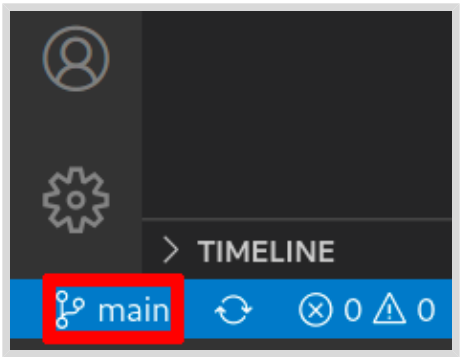


Figure 3.2: Select the main branch

Warning

Each exercise uses a unique branch. Always change to the `main` branch before you create a new branch. The new branch must use the `main` branch as the base.

    2.6. Click the **main** branch in VS Code.

Select `Create new branch...` from the list of options.

2.7. At the prompt, enter `update-app` for the branch name.

2.8. Push the `update-app` branch to your `DO101x-apps` GitHub repository.

Click the **Publish Changes** cloud icon next to the `update-app` branch name to push your local branch to your remote Git repository.

If prompted to authorize VS Code, then click **Allow** and continue on the GitHub website.

Alternatively, you can choose not to authorize GitHub. In such case, you will be prompted for username and password. Enter your GitHub developer token as your username, and leave the password field empty.

2.9. Deploy the `version` application from your `DO101x-apps` Git repository.

Switch to a command line terminal. Enter the command below in a single line with no line breaks. Do not type the \ character at the end of the lines. The command is split into multiple lines in this example for clarity and formatting purposes only:

```
$ oc new-app --name version \
    https://github.com/yourgituser/DO101x-apps#update-app \ (1)
    --context-dir version (2)
```

(1) The GitHub URL. Change *youruser* to your GitHub user. The `#update-app` indicates that RHOCP should use the `update-app` Git branch, which was created in the previous step.

(2) The `--context-dir` parameter indicates the directory under the `DO101x-apps` repository where the application source code is stored.

The preceding `oc` command creates and deploys the Node.js application. The output includes information about the latest image available for the `nodejs` image tag.

Note

This course uses the backslash character (\) to break long commands. On Linux and macOS, you can use the line breaks.

On Windows, use the backtick character (`) to break long commands.

Alternatively, do not break long commands.

3. Test the application.

3.1. Open a web browser and access the RHOCP web console. Log in to the RHOCP web console by using your developer account.

3.2. Switch to the `Developer` perspective.

3.3. Click `Topology` in the navigation pane. From the `Project:` list, select the *`yourname`*`-dev` project.

3.4. Click the `version` icon, and then click the `Details` tab. Verify that a single pod is running. You might have to wait for a minute or two for the application to be fully deployed.

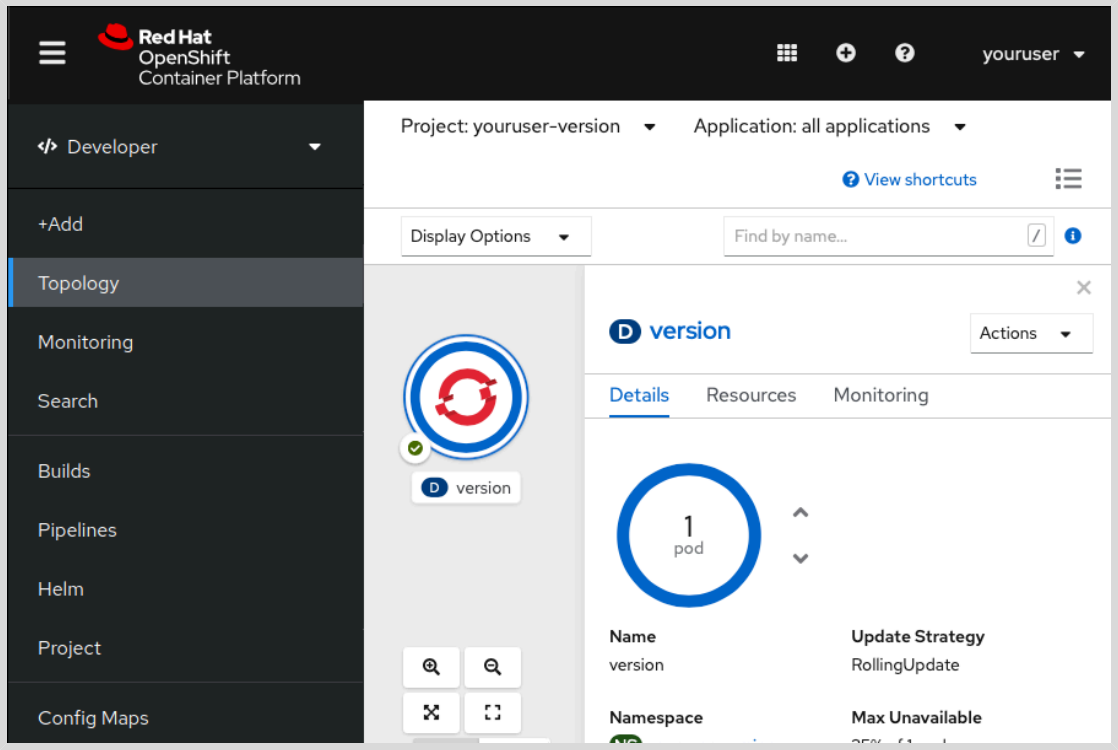


Figure 3.3: Application deployment complete

3.5. Applications created by using the `oc new-app` command do not create a route resource by default. Run the following command to create a route and allow access to the application from the internet:

```
$ oc expose svc/version
route.route.openshift.io/version exposed
```

Note

You can also create a route by using the RHOCP web console; select the `Administrator` perspective, and then click **Networking** > **Routes** > **Create Route**.

3.6. View the `Topology` page again, and observe that the `version` deployment now displays an icon to open a URL.

Click `Open URL` to view the application.

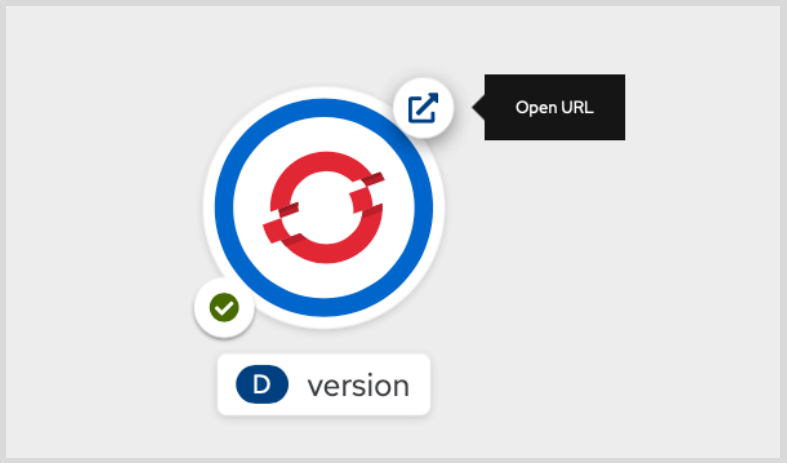


Figure 3.4: Route URL

3.7. The application opens in a new browser tab. You should see version 1 of the application as output.

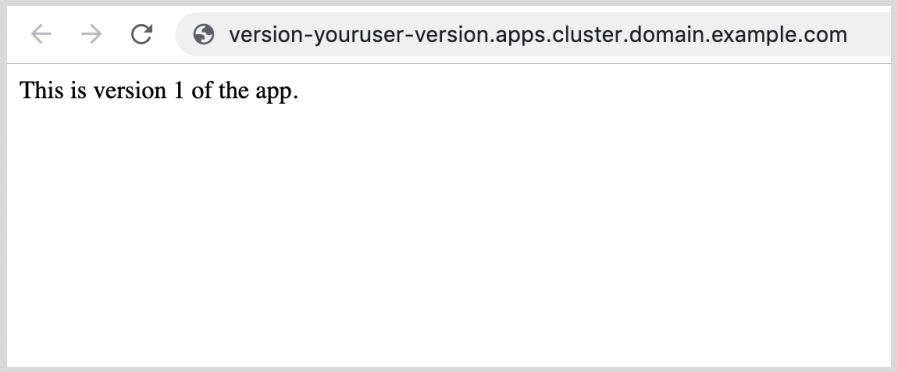


Figure 3.5: Application version 1

Close the browser tab.

4. Update the `version` application.

4.1. In the VS Code explorer view (**View** > **Explorer**), click the `app.js` file in the `version` application folder of the `DO101x-apps` folder. VS Code opens an editor tab for the `app.js` file.

4.2. Change the `response` variable to version 2 as follows:

```
...output omitted...
var response;

app.get('/', function (req, res) {
    response = 'This is version 2 of the app.' + '\n';

    //send the response to the client
    res.send(response);

});
...output omitted...
```

4.3. Save the changes to the file.

4.4. Commit your changes locally, and then push the new commit to your GitHub repository.

Switch to the `Source Control` view (**View** > **SCM**). Locate the entry for the `app.js` file under the `CHANGES` heading for the `DO101x-apps` directory.

Click the plus (+) button to add the `app.js` file changes to your next commit.

4.5. Add a commit message of `updated app to v2` in the message prompt, and then click the check mark button to commit the staged changes.

4.6. Click the **Views and more actions** > **push** to publish the changes to the remote repository.

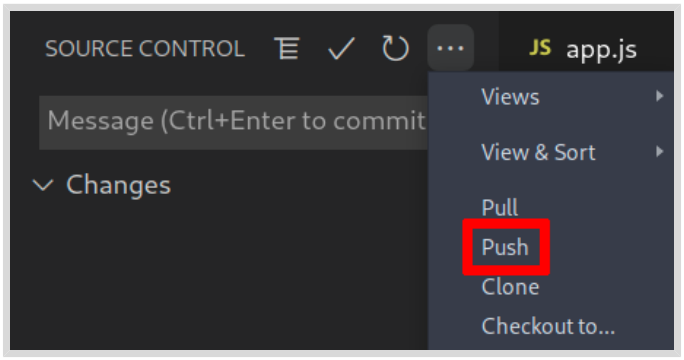


Figure 3.6: Push changes to remote repository

If prompted that this action will push and pull commits to and from the origin, then click **OK** to continue.

5. Trigger a new build manually by using the RHOCP web console.

5.1. Click **Builds** in the navigation pane to view the **Build Configs** page.

5.2. Expand the menu to the right of the `version` build config, and then click `Start Build`.

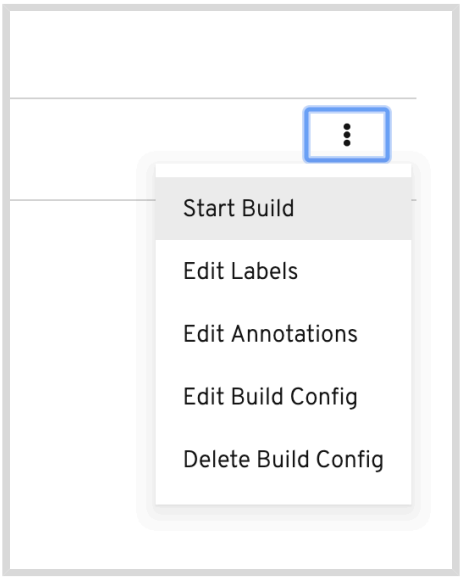


Figure 3.7: Start build

5.3. You should now see the details page for the build `version-2`.

Click the **Logs** tab to see the progress of the build. A new container image with the updated source code is built and pushed to the RHOCP image registry.

Wait until the application container image is built and deployed. You should see a `Push successful` message in the logs before proceeding to the next step.

5.4. Click on `Topology` in the navigation pane, and then click `Open URL` from the `version` deployment. Version 2 of the application is displayed.

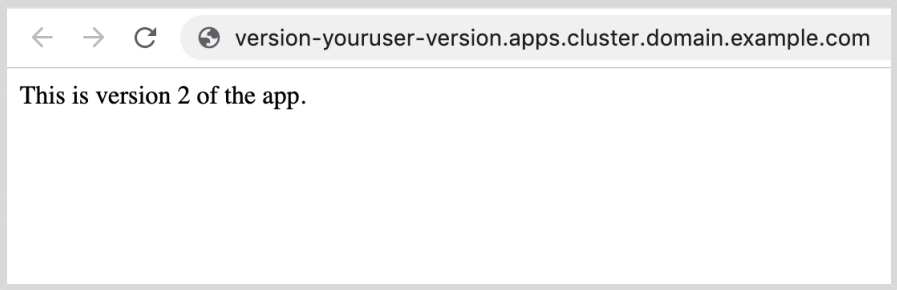


Figure 3.8: Version 2 of the application

Close the browser tab.

6. Set up webhooks in GitHub to automatically trigger a new build when you push changes to the repository.

6.1. Navigate to your `DO101x-apps` repository (https://github.com/ *yourgituser* /DO101x-apps) on GitHub using a web browser. Log in to your GitHub account if prompted.

6.2. Click the **Settings** tab to open the settings page for the repository.

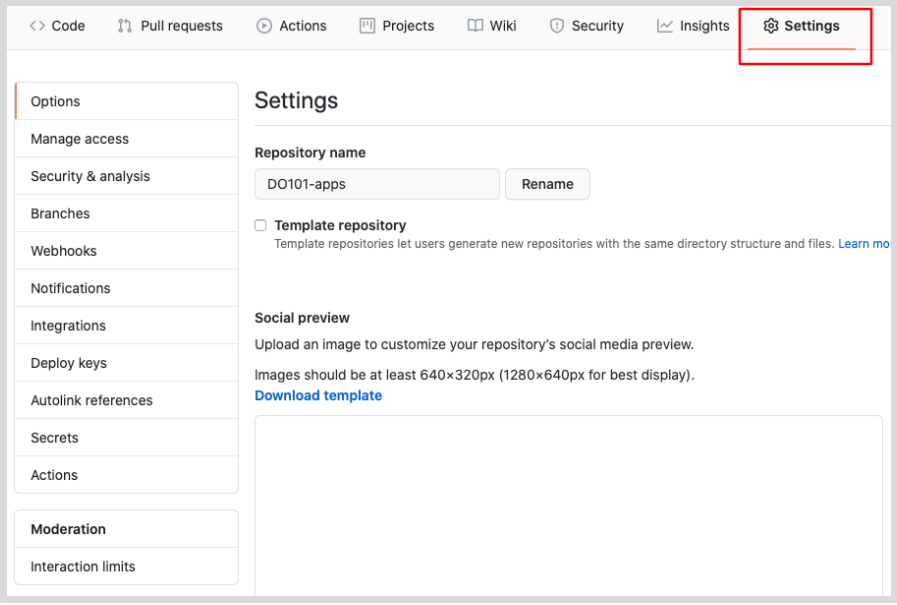


Figure 3.9: GitHub repository settings

6.3. Click **Webhooks** in the left menu to open the webhooks page.

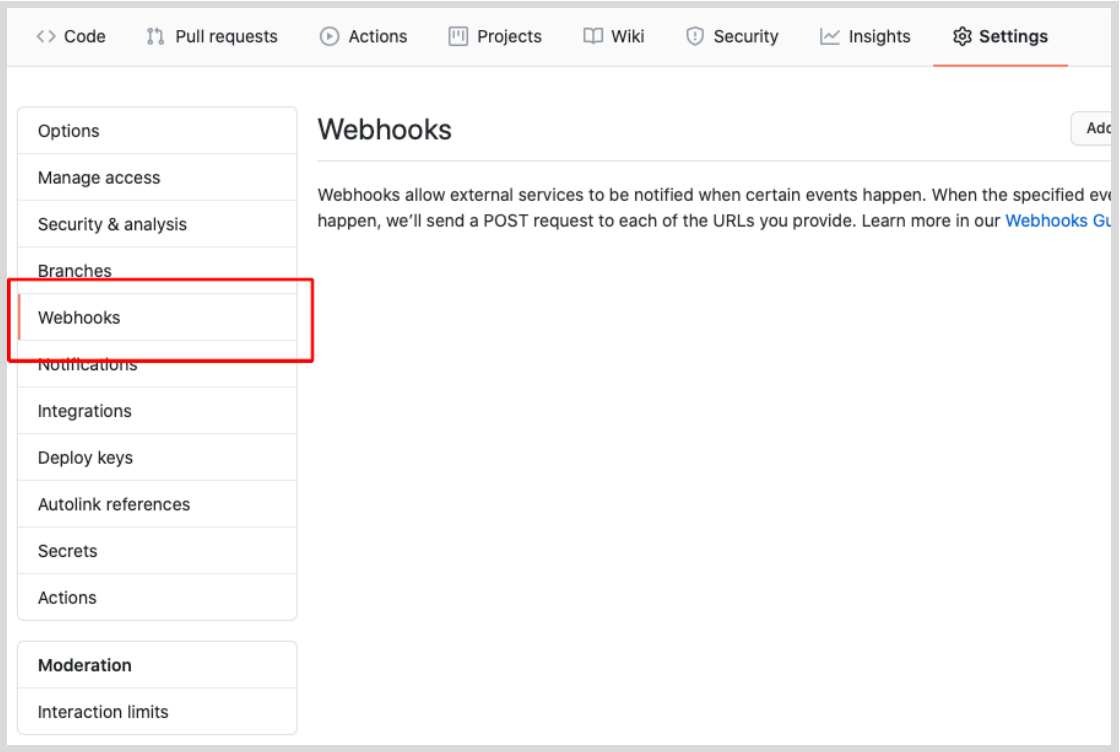


Figure 3.10: Webhooks page

6.4. Click **Add webhook** to add a new webhook. You may be asked to reenter your GitHub password. After your password is confirmed, the `Add Webhook` page displays.

6.5. Get the webhook payload URL.

Switch to the RHOCP web console in your browser, and then click `Builds` in the navigation pane.

6.6. Click the **version** build config to bring up the `Build Config Details` page.

Scroll to the bottom of this page and click the `Copy URL with Secret` link next to the `GitHub` type.

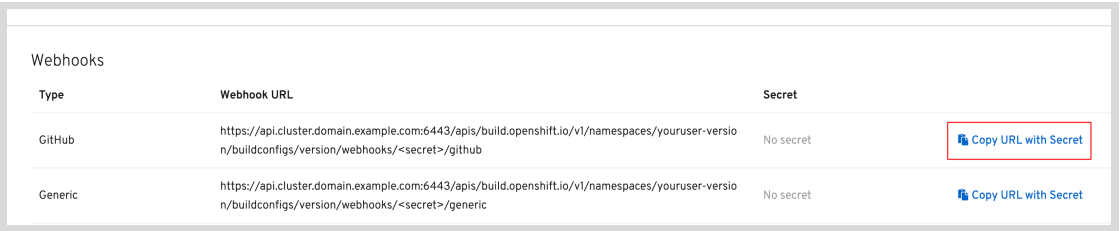


Figure 3.11: Payload URL

6.7. Switch back to the GitHub webhooks page.

Paste the payload URL into the `Payload URL` field.

Change the `Content type` to `application/json`.

Do not change any other fields.

Click **Add webhook** to add the webhook.

GitHub sends a test request to the payload URL to verify availability, and displays a green check mark for the webhook, if it was successful.

Refresh the page to see the checkmark.

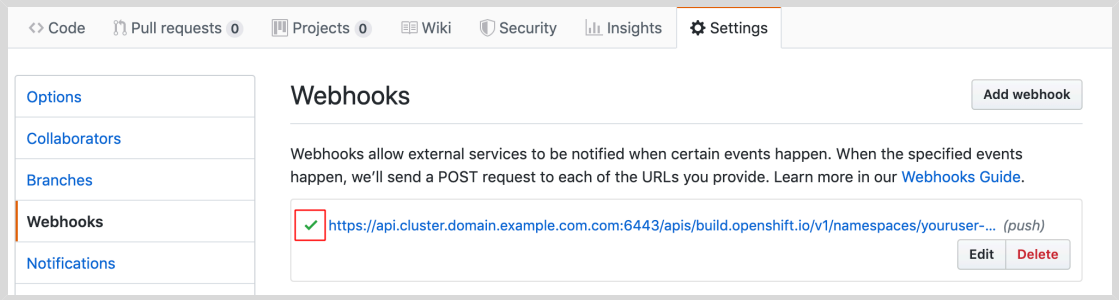


Figure 3.12: Webhook active

7. Push changes to the Git repository to automatically trigger a new build.

7.1. Edit the `DO101x-apps/version/app.js` file in VS Code to change the `response` variable to version 3, as follows:

```
...output omitted...
var response;

app.get('/', function (req, res) {
    response = 'This is version 3 of the app.' + '\n';

    //send the response to the client
    res.send(response);

});
...output omitted...
```

7.2. Save the changes to your file.

7.3. Stage the changes, commit, and then push the changes to your Git repository using steps similar those used for version 2 of the application.

7.4. After you commit and push the changes to the remote Git repository, RHOCP starts a new build.

In the RHOCP web console, click **Builds** > **version**, and then click the `Builds` tab to view the list of builds for the `version` application. Notice that a new build, `version-3` has started.

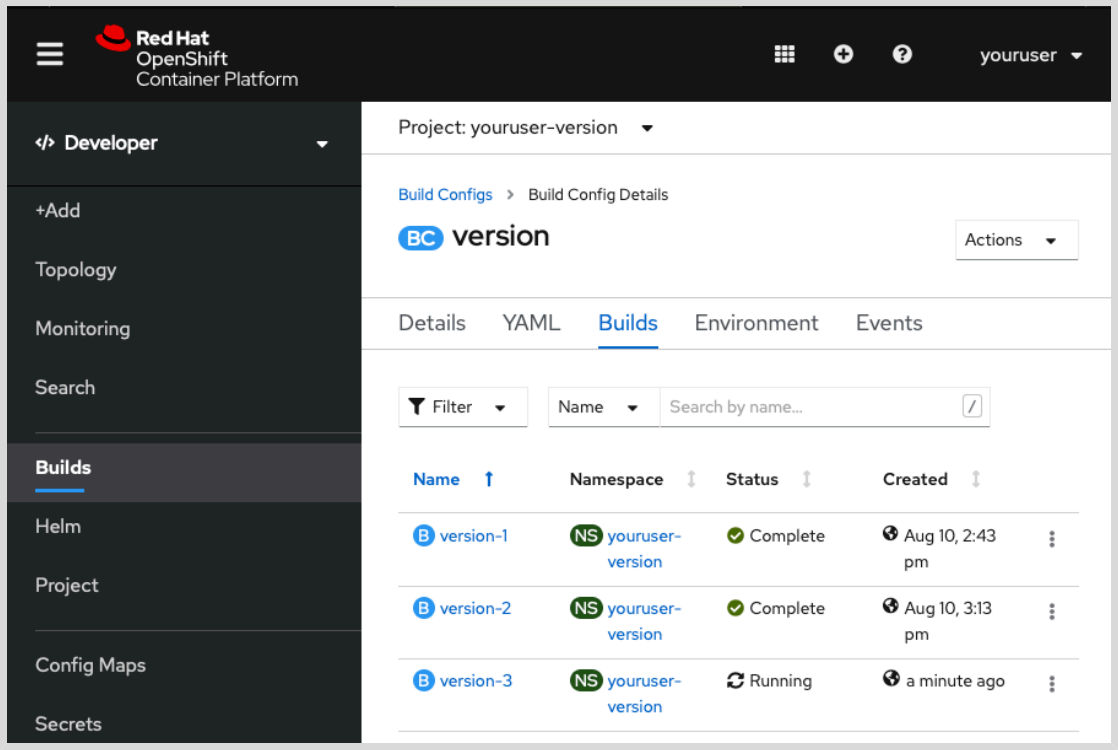


Figure 3.13: Build list

7.5. Wait for the build to complete successfully, and for the application to be redeployed.

Click `Topology`, and then click `Open URL` to view the updated application. It should now display version 3 in the output.

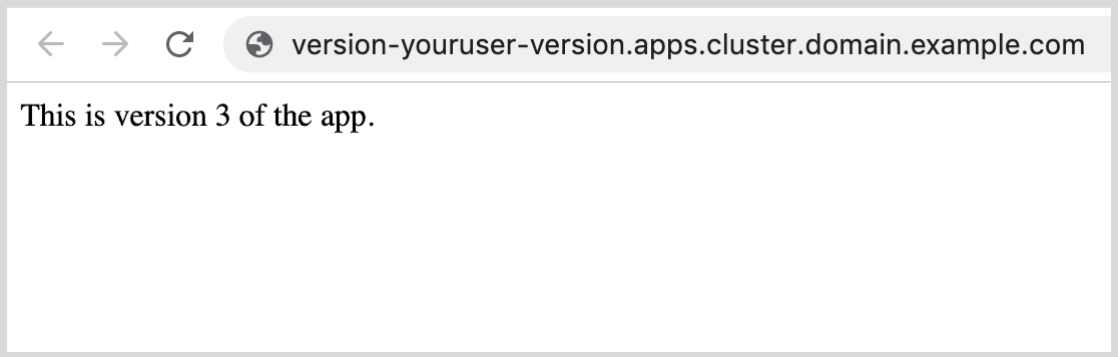


Figure 3.14: Version 3 of app

Close the browser tab.

8. Clean up.

Delete the *`youruser`*`-version` project and remove the webhooks from GitHub.

8.1. Navigate to the `Settings` page of the `DO101x-apps` repository in GitHub.

Click **Webhooks** to view the webhooks page.

8.2. Click **Delete** next to the active webhook, and then confirm the deletion in the resulting confirmation window. You might be prompted for your GitHub password.

8.3. List all RHOCP objects you created:

```
$ oc get all --selector app=version
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
service/version   ClusterIP   172.30.198.42   <none>        8080/TCP   8m44s

NAME                      READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/version   1/1     1            1           8m45s
...output omitted...
```

8.4. Delete RHOCP objects with the `app=version` label:

```
$ oc delete all --selector app=version
service "version" deleted
deployment.apps "version" deleted
buildconfig.build.openshift.io "version" deleted
imagestream.image.openshift.io "version" deleted
route.route.openshift.io "version" deleted
```

Finish

This concludes the guided exercise.

Configuring Application Secrets

Objectives

After completing this section, you should be able to add and adjust application configuration and secrets for applications deployed on Red Hat OpenShift Container Platform.

Externalizing Application Configuration in OpenShift

Cloud-native applications store application configuration as environment variables rather than coding the configuration values directly in the application source code. The advantage of this approach is that it creates a separation between the application configuration and the environment in which the application is running. Configuration usually varies from one environment to another, whereas source code does not.

For example, suppose you want to promote an application from a development environment to a production environment, with intermediate stages such as testing and user acceptance. The source code remains the same, but the configuration details specific to each environment, such as connection details to a non-production database, must not be static and must be managed separately.

Configuring Applications Using Secrets and Configuration Maps

Red Hat OpenShift Container Platform provides `Secret` and `Configuration Map` resources to externalize configuration for an application.

Secrets are used to store sensitive information, such as passwords, database credentials, and any other data that should not be stored in clear text.

Configuration maps, also commonly called `config maps`, are used to store non-sensitive application configuration data in clear text.

You can store data in secrets and configuration maps as key-value pairs or you can store an entire file (for example, configuration files) in the secret. Secret data is base64 encoded and stored on disk, while configuration maps are stored in clear text. This provides an added layer of security to secrets to ensure that sensitive data is not stored in plain text that humans can read.

Warning

Data encoded in the base64 format is not cryptographically safe. A third party can decode the base64 format. Additionally, the values in the Secret objects can be stored in the plain-text form in the `etcd` data store.

Do not use Secret objects to store sensitive data like production database passwords.

The following is an example configuration map definition in YAML:

```
kind: ConfigMap (1)
apiVersion: v1
data: (2)
    username: myuser
    password: mypass
metadata:
    name: myconf (3)
```

(1) OpenShift resource type (configuration map)

(2) Data stored in the configuration map

(3) Name for the configuration map

The following is an example secret definition in YAML:

```
kind: Secret (1)
apiVersion: v1
data: (2)
    username: cm9vdAo=
    password: c2VjcmV0Cg==
metadata:
    name: mysecret (3)
    type: Opaque
```

(1) OpenShift resource type (secret)

(2) Data stored in the secret in base64 encoded format

(3) Name for the secret

Note that the data in the secret (`username` and `password`) is encoded in base64 format, and not stored in plain text like the configuration map data.

Note

You can decode the values into plain text.

For example, on a Linux system, decode the `c2VjcmV0Cg==` value:

```
$ echo "c2VjcmV0Cg==" | base64 --decode
secret
```

After creating secrets and configuration maps, you must associate the resources with applications by referencing them in the deployment configuration for each application.

OpenShift automatically redeploys the application and makes the data available to the application.

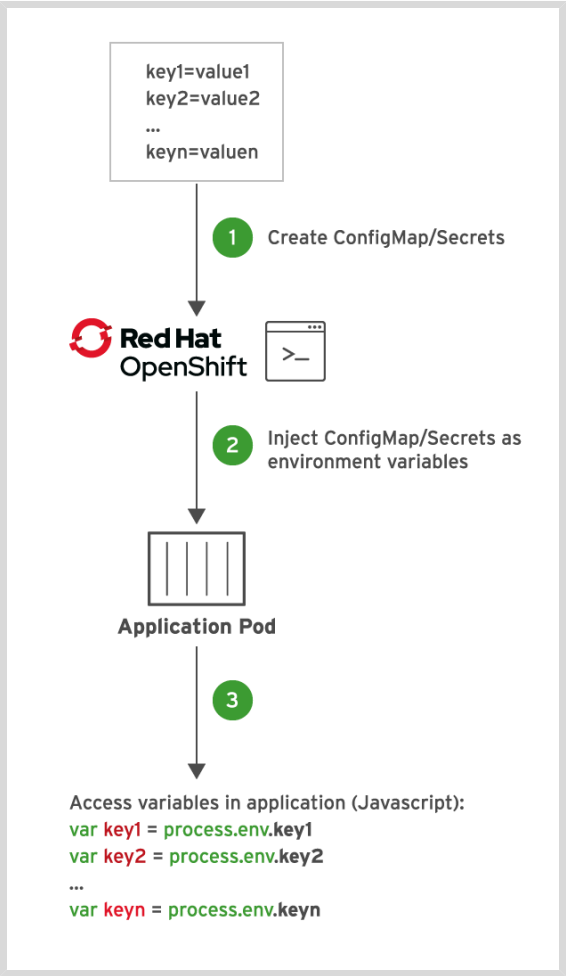


Figure 3.15: Configuration maps and secrets

The workflow for creating and using secrets and configuration maps on OpenShift is as follows:

1. Create configuration maps and secrets by using the OpenShift web console, or the OpenShift command line client (`oc`). You can store key-value pairs, or an entire file.

2. After you edit the deployment configuration for the application, and map the environment variables to use the secrets and configuration maps configured in the project, OpenShift injects the secrets and configuration maps into application pods.

3. The application accesses the values at run time, by using key based look-ups. OpenShift converts the base64 encoded data into the format that the application can read.

In addition to exposing secrets and configuration maps as environment variables, you can also expose them as files. This is useful when you store entire configuration files in secrets or configuration maps.

For example, if your application configuration is in an XML file, then you can store the file in a configuration map or secret and mount it inside the container, where your application can then parse it at start-up. To make changes to the configuration, you edit the configuration map or secret, and then OpenShift automatically redeploys the application container and picks up the changes. You do not need to rebuild your application or the container image.

Note that if you choose to mount the secrets and configuration maps as files inside the application pods, OpenShift mounts the file using a read-only temporary file system, with a file for each key, and the key value as the content of the corresponding file.

When the value of a secret or a configuration map changes, you must restart all pods that use the values for the new value to propagate into the pod.

References

OpenShift Secrets
https://docs.openshift.com/container-platform/4.6/nodes/pods/nodes-pods-secrets.html

Configuring your application in OpenShift
https://blog.openshift.com/configuring-your-application-part-1/

Guided Exercise

Configuring Application Secrets

In this exercise, you will deploy a Node.js application that uses configuration maps and secrets on Red Hat OpenShift Container Platform.

Outcomes

You should be able to:

- Create configuration maps and secrets by using the OpenShift web console.
- Update the deployment configuration for an application to use the configuration maps and secrets.
- Deploy an application that uses the configuration maps and secrets.

Before You Begin

To perform this exercise, ensure you have access to:

- A running Red Hat OpenShift (RHOCP) cluster. See *Guided Exercise: Deploying an Application to Red Hat OpenShift Container Platform* for more information about using the developer sandbox RHOCP cluster.
- The source code for the `weather` application in the `DO101x-apps` Git repository on your local system.
- The `oc` command.

Instructions

1. In this exercise, you will use the OpenWeatherMap API to fetch the weather forecast for cities around the world. To invoke the OpenWeatherMap API, you need a unique API key.

1.1. Create a new account for the OpenWeatherMap API. Navigate to the website https://openweathermap.org/ in a web browser.

1.2. To create a new account, click `Sign in` and then `Create an Account`.

1.3. Enter a `username`, `email`, and `password` for your account. Select the check boxes to confirm your age, and agree to the terms and conditions.

Do not select any of the three options related to receiving communications from OpenWeather. Select the `I'm not a robot` option, and then click **Create Account**.

1.4. When asked to provide details about the usage of the API, enter your name and select `Other` in the `Purpose` field.

1.5. After you have logged in, click `Services` to view the services offered for your free account. There are restrictions on free accounts that limit the number of API calls you can make in a given duration.

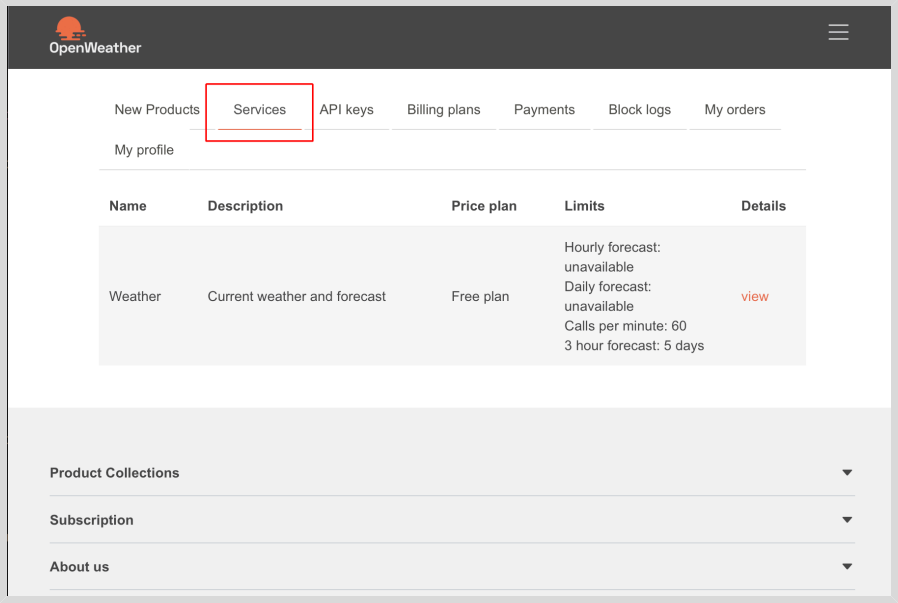


Figure 3.16: API services

1.6. Click `API keys` to view the API keys for your account.

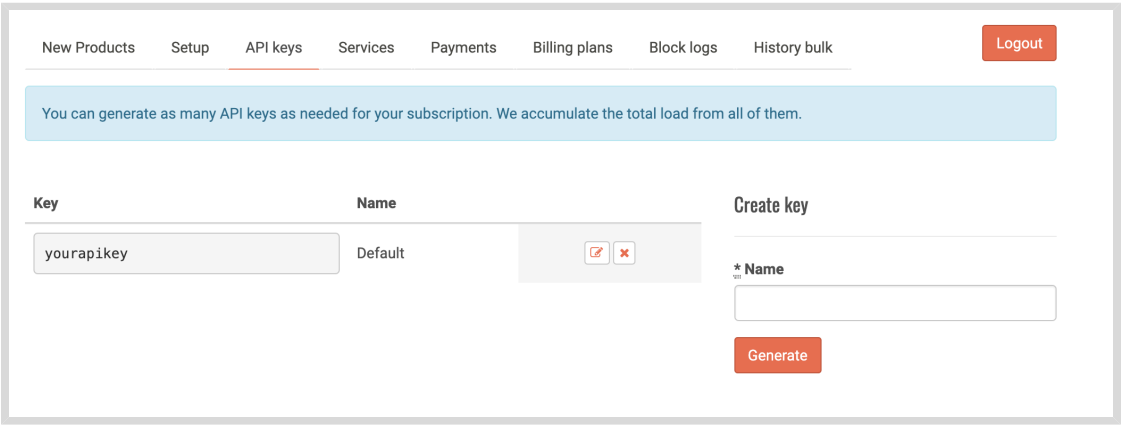


Figure 3.17: API keys

1.7. A default API key is generated for your account. Copy this API key to a temporary file, or keep the browser tab open. You need the API key to create an OpenShift secret for the `weather` application. You can also generate more keys by clicking **Generate**.

To verify that your API key is active, invoke the URL `http://api.openweathermap.org/data/2.5/weather?q=London&appid=`*`api_key`* in a browser.

Important

It may take a few minutes for your API key to be activated before you can invoke the OpenWeatherMap API.

Replace `api_key` with your default API key from the previous step. If the API key is activated, you will get a JSON response like the following with forecast data for London:

```
{"coord":{"lon":-0.13,"lat":51.51},"weather": ...output omitted...
{"temp": ...output omitted...
{"type":1,"id":1414,"country":"GB","sunrise": ...output omitted...
```

If the key is not yet activated, you will get a code 401 `Invalid API key` message. Continue with the next steps in the exercise while the API key gets activated, and retest it after the application is deployed in OpenShift.

2. Inspect the source code for the `weather` application.

2.1. Launch the Visual Studio Code (VS Code) editor, and then open the `DO101x-apps` folder in the `My Projects` workspace. The source code for the `weather` application is in the `weather` directory.

2.2. Inspect the `DO101x-apps/weather/package.json` file to view the package dependencies for this Node.js application.

The `weather` application is a simple web application that is based on the popular *Express.js* framework. The `weather` application uses the `node-fetch` HTTP client package to access the OpenWeatherMap API and display weather forecasts for numerous cities around the world.

```
"dependencies": {
  "cookie-parser": "1.4.5",
  "debug": "4.3.2",
  "dotenv": "10.0.0",
  "express": "4.17.1",
  "http-errors": "1.8.0",
  "morgan": "1.10.0",
  "node-fetch": "^2.6.1",
  "package-json": "^7.0.0",
  "pug": "3.0.2"
}
```

2.3. Inspect the `DO101x-apps/weather/app.js` file, which is the main entry point for the application. There is a single Express.js route definition called `indexRouter`:

```
...output omitted...
var cookieParser = require('cookie-parser');
var logger = require('morgan');

app.use('/', indexRouter);
var app = express();
...output omitted...
```

2.4. The code for the `indexRouter` route is defined in the `DO101x-apps/weather/routes/index.js` file. Open this file in VS Code. This file contains the main business logic in the application.

2.5. The first method handles HTTP GET requests to the '/' URL:

```
...output omitted...
router.get('/', function(req, res) {
  res.render('index', { weather: null, err: null });
});
...output omitted...
```

All HTTP GET requests to the '/' URL are redirected to a page with an HTML form that enables you to enter a city name for which you want the weather forecast.

Note

The code for the HTML form is in the `DO101x-apps/weather/views/index.pug` file.

2.6. The second method handles HTTP POST requests from the HTML form by invoking the OpenWeatherMap API, and then passes the resulting JSON response to the HTML front end:

```
...output omitted...
router.post('/get_weather', async function (req,res) {
  let city = req.body.city;
  let url = http://api.openweathermap.org/data/2.5/weather?q=${city}&units=
${UNITS}&appid=${OWM_API_KEY};

  try {
    let data = await fetch(url);
    let weather = await data.json();
...output omitted...
```

2.7. Note that invoking the OpenWeatherMap API requires an API key. The API key is injected as an environment variable at run time. You will create an OpenShift secret to store the API key:

```
...output omitted...
const OWM_API_KEY = process.env.OWM_API_KEY || 'invalid_key';
...output omitted...
```

2.8. You will also create a configuration map to store application specific configuration in plain text. The `UNITS` environment variable controls if the weather forecast is displayed in metric (degree celsius), or imperial units (fahrenheit):

```
...output omitted...
const UNITS = process.env.UNITS || 'metric';
...output omitted...
```

3. Create a new branch in your Git repository for the `weather` application.

3.1. Ensure that the `DO101x-apps` repository uses the `main` branch. If you were working with another branch for a different exercise, click the current branch and then select `main` in the `Select a ref to checkout` window to switch to the `main` branch.

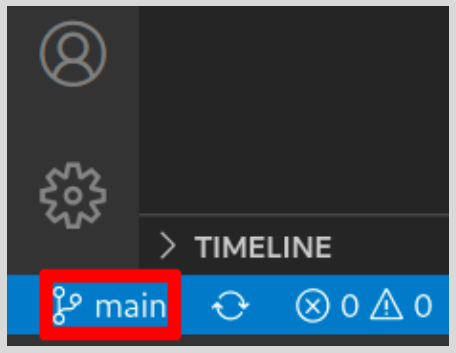


Figure 3.18: Select the main branch

Warning
Each exercise uses a unique branch. Always change to the `main` branch before you create a new branch. The new branch must use the `main` branch as the base.

3.2. Click the **main** branch in VS Code.

Select `Create new branch...` from the list of options.

3.3. At the prompt, enter `weather` for the branch name.

The Source Control view updates the `DO101x-apps` entry with the new branch name.

3.4. Push the `weather` branch to your `DO101x-apps` GitHub repository.

Click the **Publish Changes** cloud icon next to the `weather` branch to push your local branch to your remote Git repository.

If prompted to authorize VS Code to push commits, then click **Allow** and continue on the GitHub website.

Alternatively, you can choose not to authorize GitHub. In such case, you will be prompted for username and password. Enter your GitHub developer token as your username, and leave the password field empty.

4. Create a secret to store the API key for the OpenWeatherMap service.

4.1. Log in to the RHOCP web console. Confirm the `Developer` perspective is active.

4.2. Click **Secrets** in the left menu.

4.3. Click **Create** > **Key/Value Secret** to create a new secret.

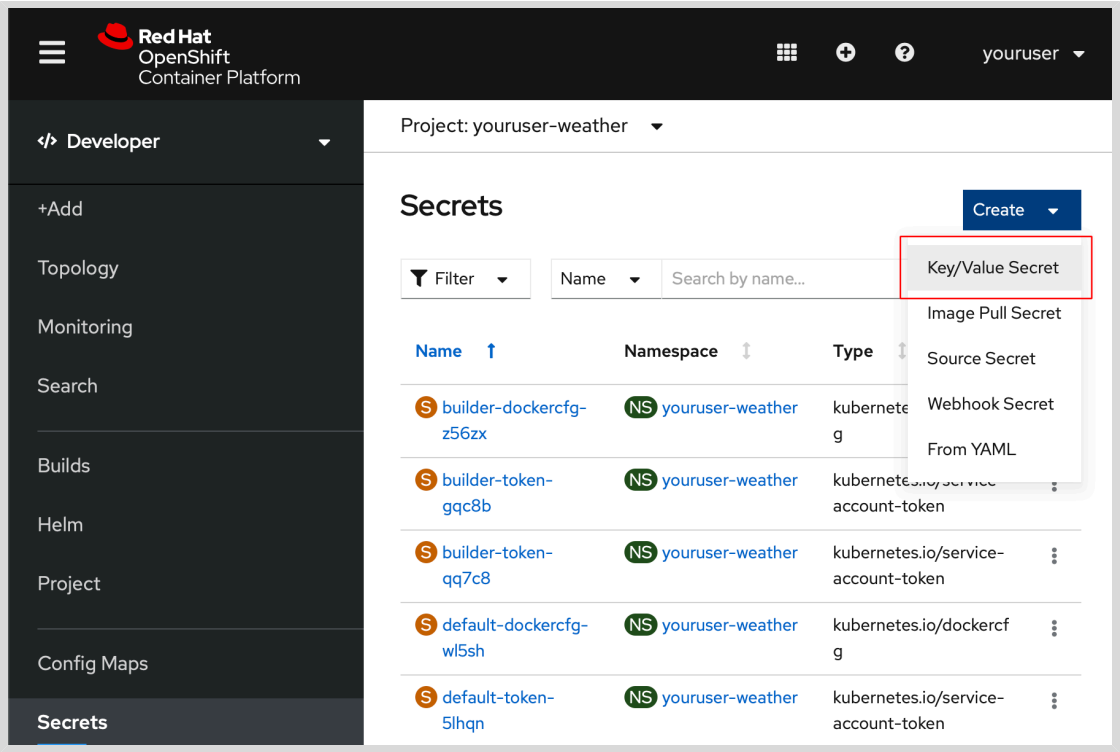


Figure 3.19: Create secret

4.4. On the `Create Key/Value Secret` page, enter `owm-api-secret` in the `Secret Name` field, `OWM_API_KEY` in the `Key` field, and then copy the default API key that was generated for your OpenWeatherMap API account to the `Value` field.

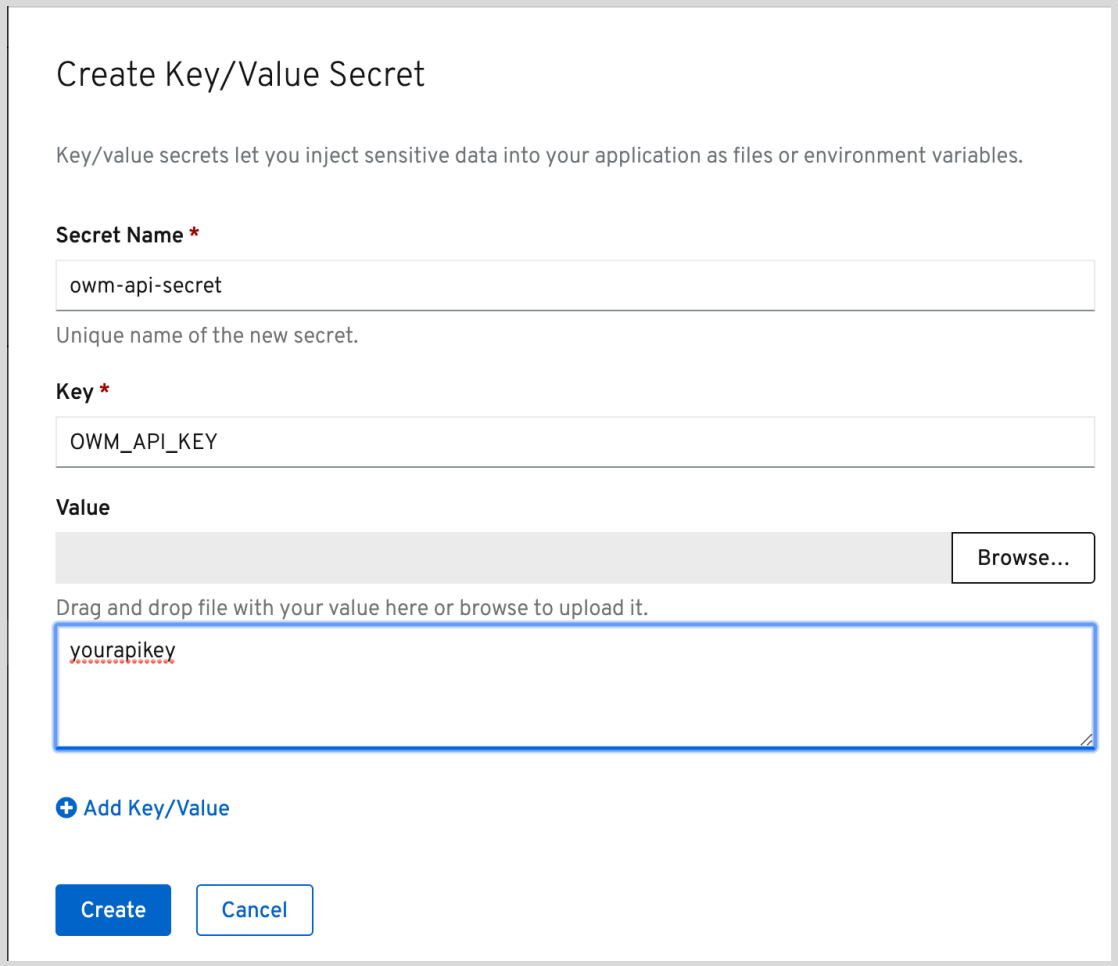


Figure 3.20: Secret details

Click **Create** to create the secret.

5. Create a configuration map to store the configuration related to the `weather` application.

   5.1. For the `weather` application, create a configuration map to hold a variable that indicates if the weather forecast should be in imperial (Fahrenheit) or metric units (Celsius).

   Select **Config Maps** from the left menu.

   5.2. Click **Create Config Map** to create a new configuration map.

   The `Create Config Map` page shows a YAML editor and sample code to create key value pairs.

   5.3. Change the name to `weather-config` and replace the data field with the `UNITS: metric` key-value pair:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: weather-config
  namespace: youruser-dev
data:
  UNITS: metric
```

Important

Ensure that you maintain correct indentation as shown in the snippet. YAML files are indentation sensitive. Two spaces precede the `UNITS` key.

Click **Create** to create the configuration map.

The snippet creates a configuration map named `weather-config` in the *youruser*`-weather` project. The configuration map stores a single variable (key) named `UNITS` with a string value `metric`.

6. Deploy the `weather` application to OpenShift.

6.1. Select **Add** in the left menu, and then click **All services** in the **Developer Catalog** section.

6.2. Click **Languages** > **JavaScript** and then click the `Builder Image` option named `Node.js`.

Warning

If you do not see the `Node.js` builder image, make sure the `Builder Image` filter is selected in the **Type** section.

Click **Create Application** to enter the details of the application.

6.3. Configure the application build according to the following table. To access the Git parameters, click **Show Advanced Git Options**.

New Application Parameters

Form Field	Value
`Git Repo URL`	`https://github.com/`*yourgituser*`/DO101x-apps`
`Git Reference`	`weather`
`Context Dir`	`/weather`
`Application Name`	`weather`
`Name`	`weather`

Do not click `Create` yet. You must first customize the deployment.

6.4. Click `Deployment` to customize the deployment options. Reference the secret and configuration map that you created earlier.

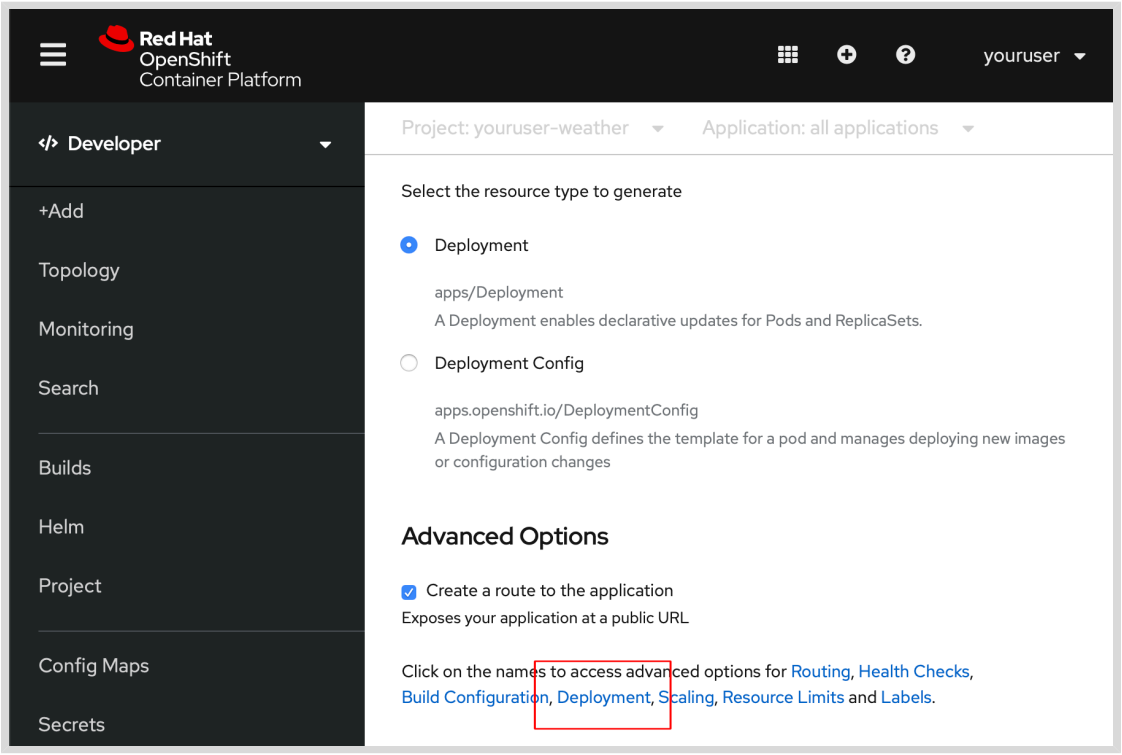


Figure 3.21: Deployment for weather application

Click **Add from Config Map or Secret**

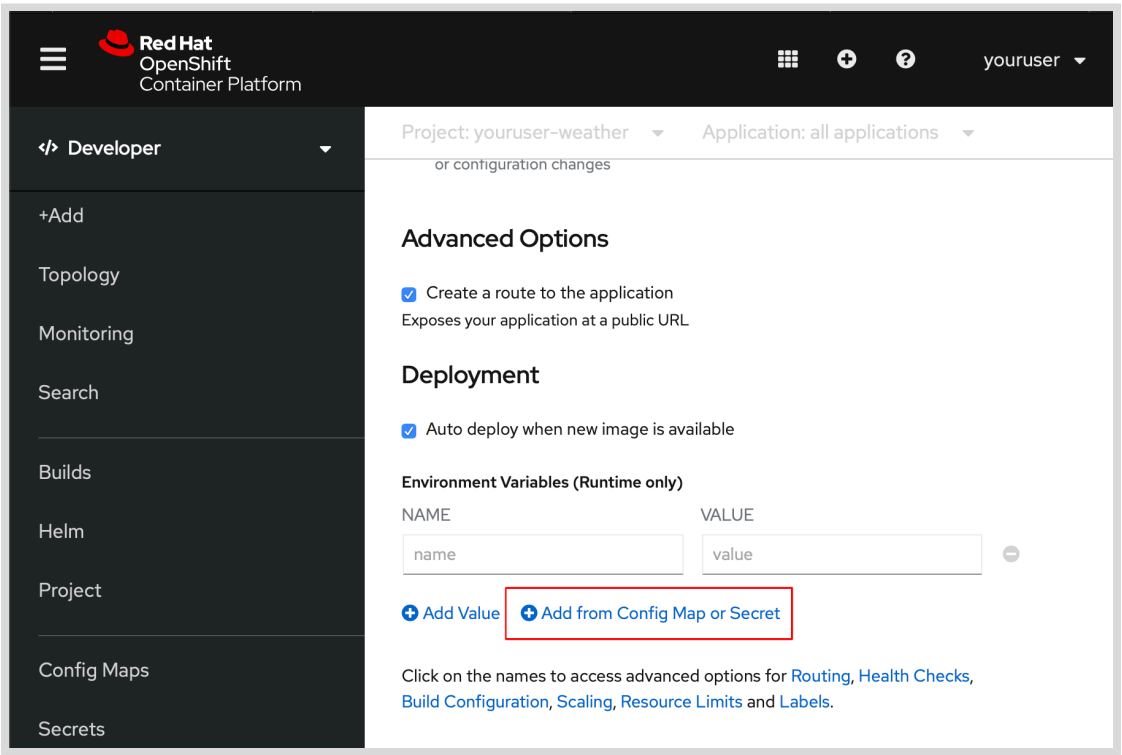


Figure 3.22: Add environment variables from config map or secret

6.5. A new row is added to the `Environment Variables` table. Add a key called `OWM_API_KEY`. Click the `Select a resource` list and select `owm-api-secret`. Click the `Select a key` list and select `OWM_API_KEY`.

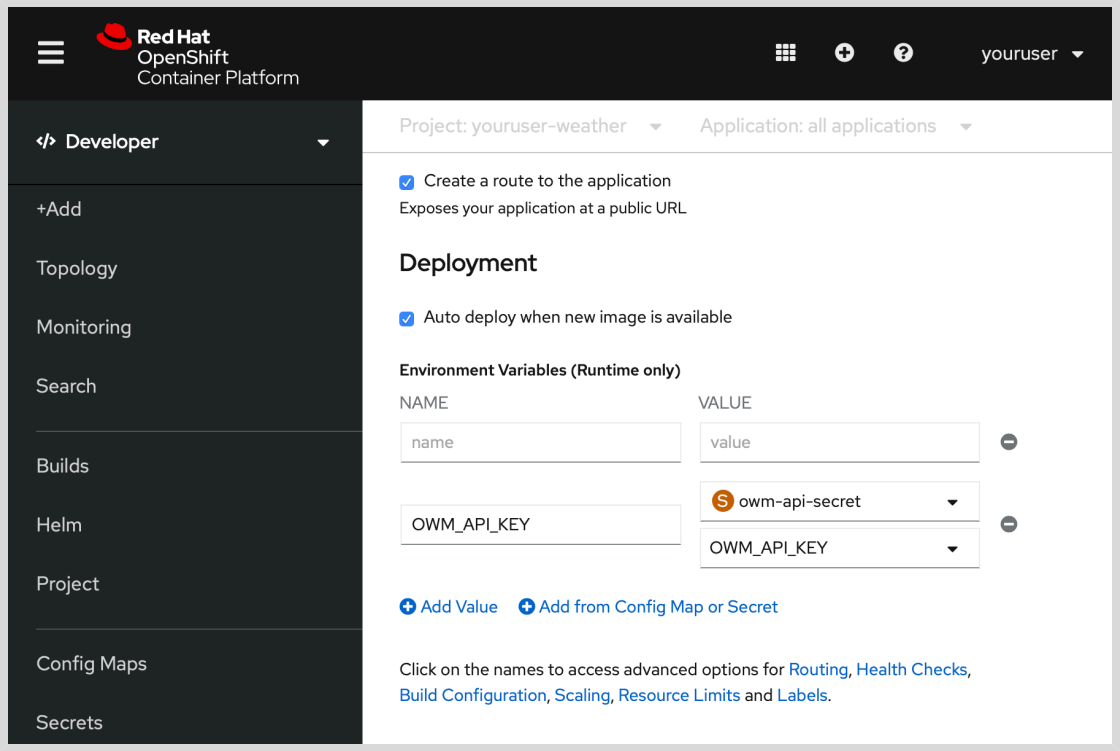


Figure 3.23: Select secret

6.6. Click **Add from Config Map or Secret** again. A third row is added to the `Environment Variables` table.

Add a key called `UNITS`. Click the `Select a resource` list and select `weather-config`. Click the `Select a key` list and select `UNITS`.

6.7. Remove the first row in the `Environment Variables` table by clicking the minus (-) icon next to the empty `value` field.

Your final `Environment Variables` table should display as follows:

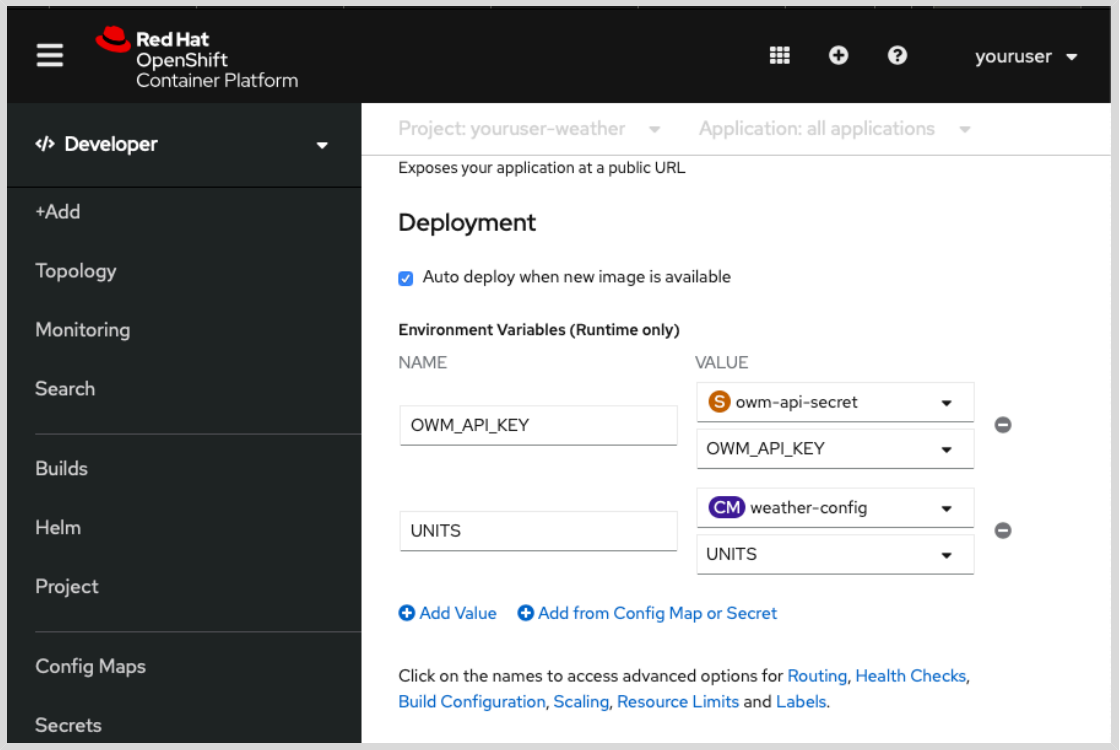


Figure 3.24: Final environment variables table

6.8. To avoid deployment errors in the following steps, review the values you entered in the form before proceeding. Click **Create** to start the build and deployment processes.

7. Test the application.

7.1. Wait for a few minutes while the application container image is built and deployed. When deployment is complete, a green tick mark displays for the `weather` deployment on the `Topology` page.

Figure 3.25: Weather app built successfully

7.2. Click the `Open URL` link to open the route URL for the `weather` application.

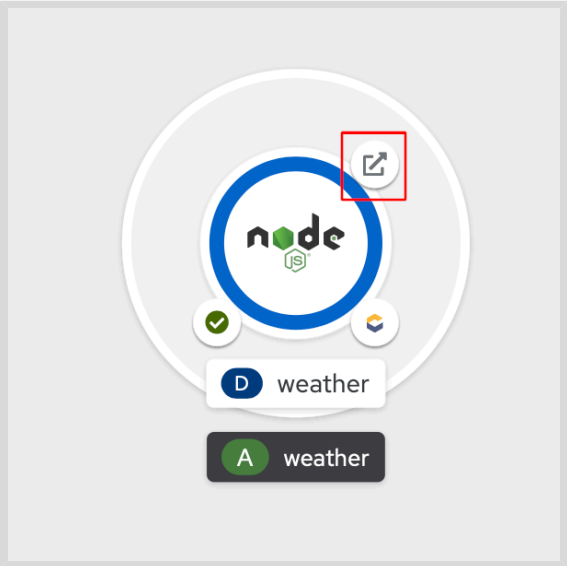


Figure 3.26: Weather route URL

7.3. The home page for the `weather` application displays.

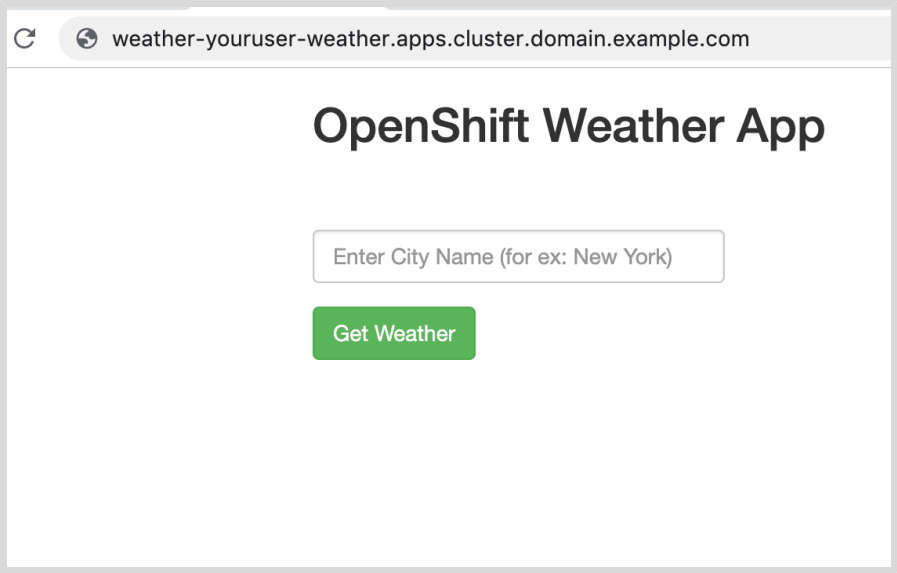


Figure 3.27: Weather app home page

7.4. Enter a city name in the field, such as "New York", and then click **Get Weather**.

7.5. The current weather forecast for the city displays in metric units.

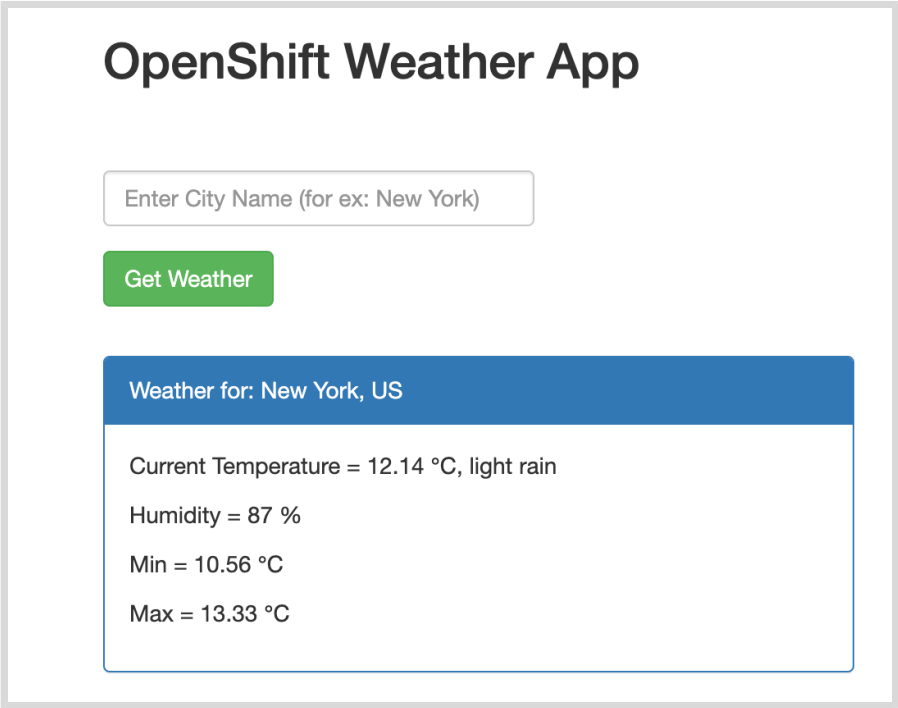


Figure 3.28: Weather forecast

Close the browser tab.

8. Switch back to the OpenShift web console to view the logs for the `weather` application.

8.1. Click `Topology`, and then click the `weather` deployment.

8.2. Click `Resources`, and then click `View Logs` for the `weather` pod to view the logs for the `weather` application.

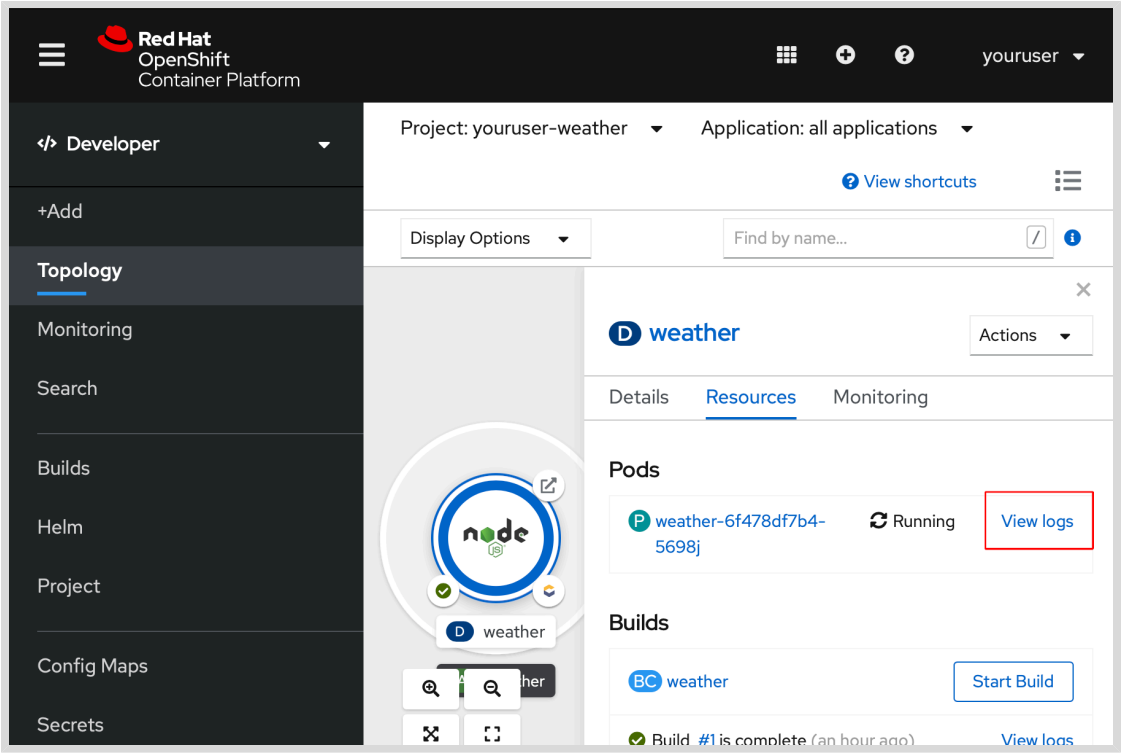


Figure 3.29: View logs for the weather application

The raw JSON response from the OpenWeatherMap API should be displayed. The application user interface filters the response and displays a subset of the data on the screen.

```
{ coord: { lon: -73.99, lat: 40.73 },
  weather:
   [ { id: 500, main: 'Rain', description: 'light rain', icon: '10d' },
     { id: 701, main: 'Mist', description: 'mist', icon: '50d' },
     { id: 300,
       main: 'Drizzle',
       description: 'light intensity drizzle',
       icon: '09d' } ],
  base: 'stations',
  main:
   { temp: 12.14,
     pressure: 1020,
     humidity: 87,
     temp_min: 10.56,
     temp_max: 13.33 },
  visibility: 16093,
  wind: { speed: 6.2, deg: 70 },
```

Figure 3.30: Weather application logs

9. Update the deployment configuration for the `weather` application to change the value of the `UNITS` key in the `weather-config` configuration map to display the forecast in imperial units.

   9.1. Select `Config Maps` from the left menu.

9.2. Click the three dots to the right of `weather-config`, and then click `Edit Config Map`.

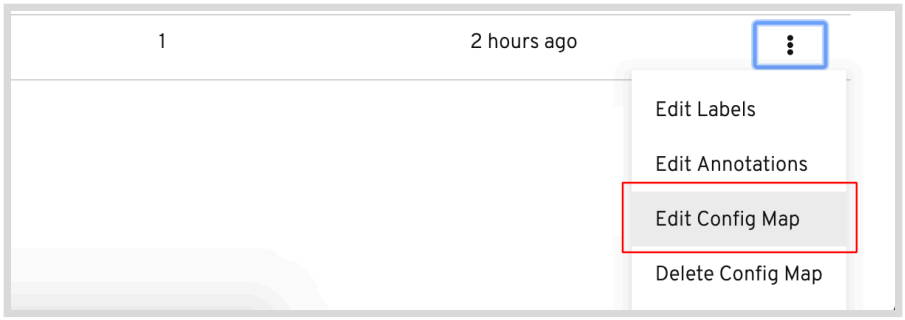


Figure 3.31: Edit config map

9.3. A page displays with an editable YAML snippet that contains the existing data in the `weather-config` configuration map.

Change the value of the `UNITS` key to `imperial` as shown below:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: weather-config
  namespace: youruser-weather
  ...output omitted...
data:
  UNITS: imperial
```

9.4. Click **Save** to save your changes.

10. Restart the application pod and verify that the weather forecast is displayed in imperial units.

10.1. Click `Topology`, and then click the `weather` deployment.

Click the pod name. Then, click **Actions** > **Delete Pod** and confirm the pod deletion. Wait for a new pod to start.

10.2. Click **Topology**. Then, click the `Open URL` link to open the route URL for the `weather` application.

10.3. Enter a city name, and then click `Get Weather`. The weather forecast displays in imperial units.

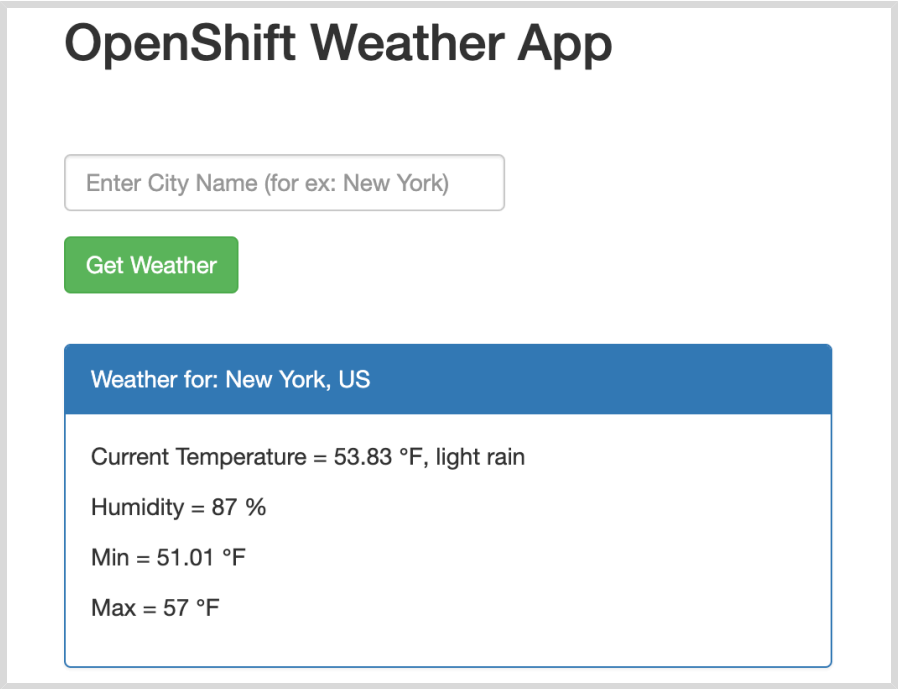


Figure 3.32: Weather forecast in imperial units

Close the browser tab.

11. Clean up.

11.1. In the RHOCP web console, click your username at the top right, and then click **Copy Login Command**. This will open a new tab and prompt you for your username and password.

11.2. Once authenticated, click **Display Token** and copy the provided `oc login` command.

11.3. Log in to your RHOCP account by using the copied command and notice the default project:

```
$ oc login --token=yourtoken --server=https://api.region.prod.nextcle.com:6443
Logged into "https://api.region.prod.nextcle.com:6443" as "youruser" using the
 token provided.
...output omitted...
Using project "youruser-dev".
```

11.4. Delete RHOCP objects with the `app=weather` label:

```
$ oc delete all --selector app=weather
pod "weather-c7867c8d9-gt6nl" deleted
service "weather" deleted
deployment.apps "weather" deleted
buildconfig.build.openshift.io "weather" deleted
imagestream.image.openshift.io "weather" deleted
route.route.openshift.io "weather" deleted
```

11.5. Delete the `weather-config` configmap object:

```
$ oc delete configmap weather-config
configmap "weather-config" deleted
```

11.6. Delete the `owm-api-secret`, `weather-generic-webhook-secret`, and `weather-github-webhook-secret` secret objects:

```
$ oc delete secret owm-api-secret weather-generic-webhook-secret
 weather-github-webhook-secret
secret "weather-generic-webhook-secret" deleted
secret "weather-github-webhook-secret" deleted
secret "owm-api-secret" deleted
```

Finish

This concludes the guided exercise.

Connecting an Application to a Database

Objectives

After completing this section, you should be able to deploy an application that connects to a database on the Red Hat OpenShift Container Platform.

Connecting to Databases

The Red Hat OpenShift Container Platform supports the deployment of a number of databases, such as MySQL, PostgreSQL, and MongoDB, by using the OpenShift web console `Developer` view, or the OpenShift command line client (`oc`).

After deploying a database, you can deploy other applications to OpenShift to access, store, and manage data in the database. Store the database credentials in an OpenShift secret, and then connect to the database from applications by using environment variables. OpenShift injects the secret data, as environment variables, into application pods at run time.

Note

When deploying a database by using one of the built-in templates provided by OpenShift, a secret containing the database user name, password, and database name is created automatically. The name of the secret is the same as the database service name.

Although you can use of this secret to connect to the database, you might want to create your own secret to store more details about the database, including application specific flags.

You can create a single secret that encapsulate all the configuration details for the database. You can safely delete the default, generated secret if you do not need it.

By externalizing the database configuration and storing it in a secret, you avoid storing sensitive information in plain text configuration files. Another advantage of this approach is support for switching between different environments, like development, staging, QA, and production, without rebuilding the application.

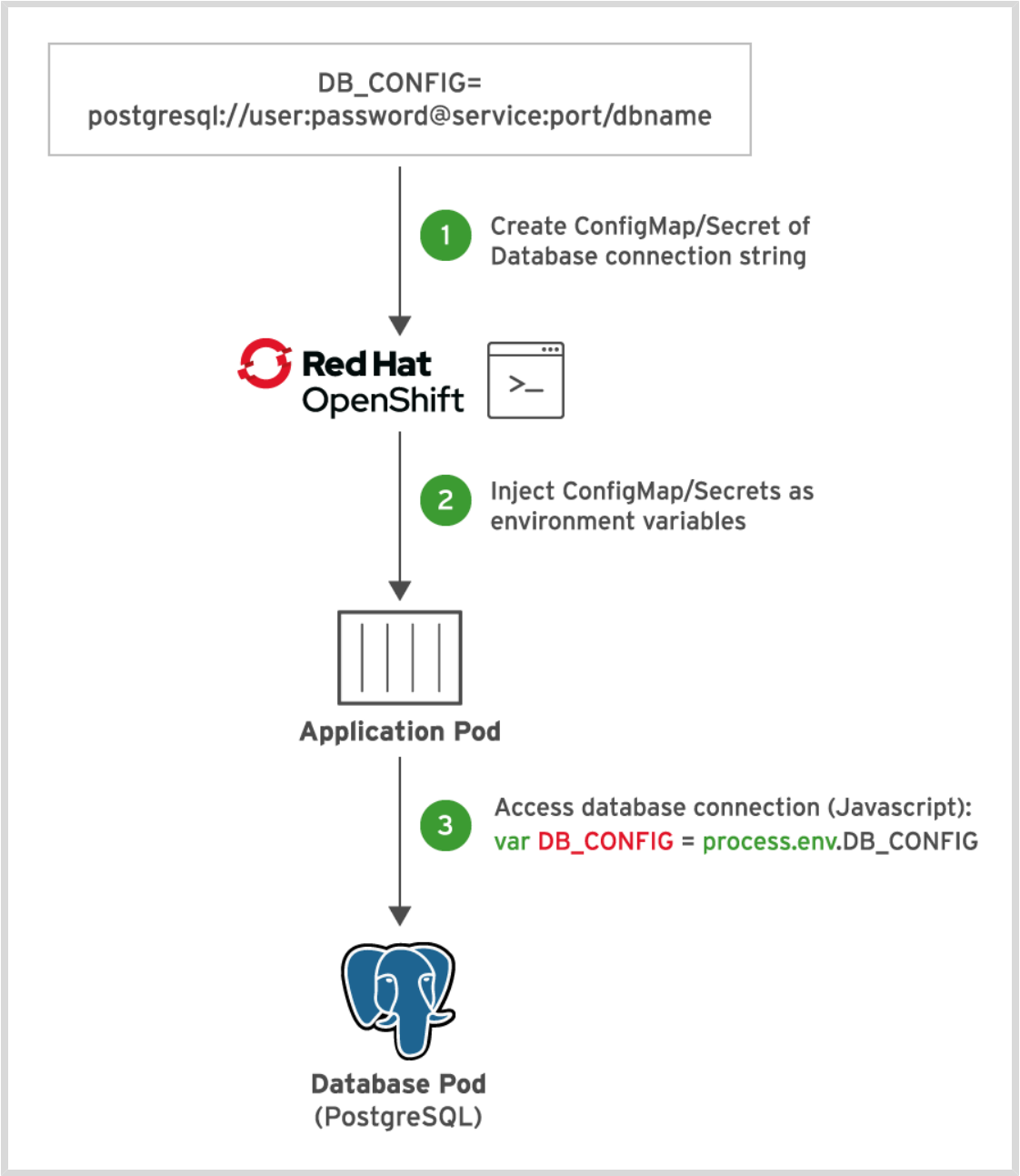


Figure 3.33: Connecting to a PostgreSQL database

The workflow for accessing databases from applications deployed on OpenShift is as follows:

1. Create a secret to store the database access configuration by using the OpenShift web console, or the OpenShift command line client (`oc`).

2. OpenShift injects the secret into application pods after you edit the deployment configuration for the application, and map the environment variables to use the secret.

3. The application accesses the values at run time, by using key based look-ups. OpenShift converts the base64 encoded data back into a format that is readable by the application.

For Node.js based JavaScript applications, the general format for the PostgreSQL database connection string is of the form:

```
postgresql://username:password@dbservice:port/dbname
```

The connection string has the following parts:

- `username` - The database user name for accessing the database
- `password` - The password for accessing the database
- `dbservice` - The service name for the database deployed on OpenShift
- `port` - The TCP port where the database server listens for incoming connections
- `dbname` - The database name

For example, deploy a PostgreSQL database on OpenShift as shown in following table:

PostgreSQL Database Details

Form Field	Value
Database Service Name	`mydbservice`
PostgreSQL Connection Username	`myapp`
PostgreSQL Connection Password	`mypass`
Port	`5432`
PostgreSQL Database Name	`mydb`

The resulting database connection string for Node.js based applications is:

```
postgresql://myapp:mypass@mydbservice:5432/mydb
```

In scenarios where the Node.js application is deployed on OpenShift, but the database is external to the cluster, the database connection string remains the same; the one exception is that the database service name is replaced by the hostname or IP address of the external database server.

For example, if your PostgreSQL database server is running on a server called `mydbhost.example.com`, then the database connection string (assuming all other details are similar to values listed in the preceding table) becomes:

```
postgresql://myapp:mypass@mydbhost.example.com:5432/mydb
```

To create a secret with the database connection string as data, and to access it from a Node.js application, use the following steps:

1. Select `Secrets` in the left menu.
2. Click **Create** > **Key/Value Secret** to create a new secret.
3. Create a new key-value secret by using the database connection string as the value.

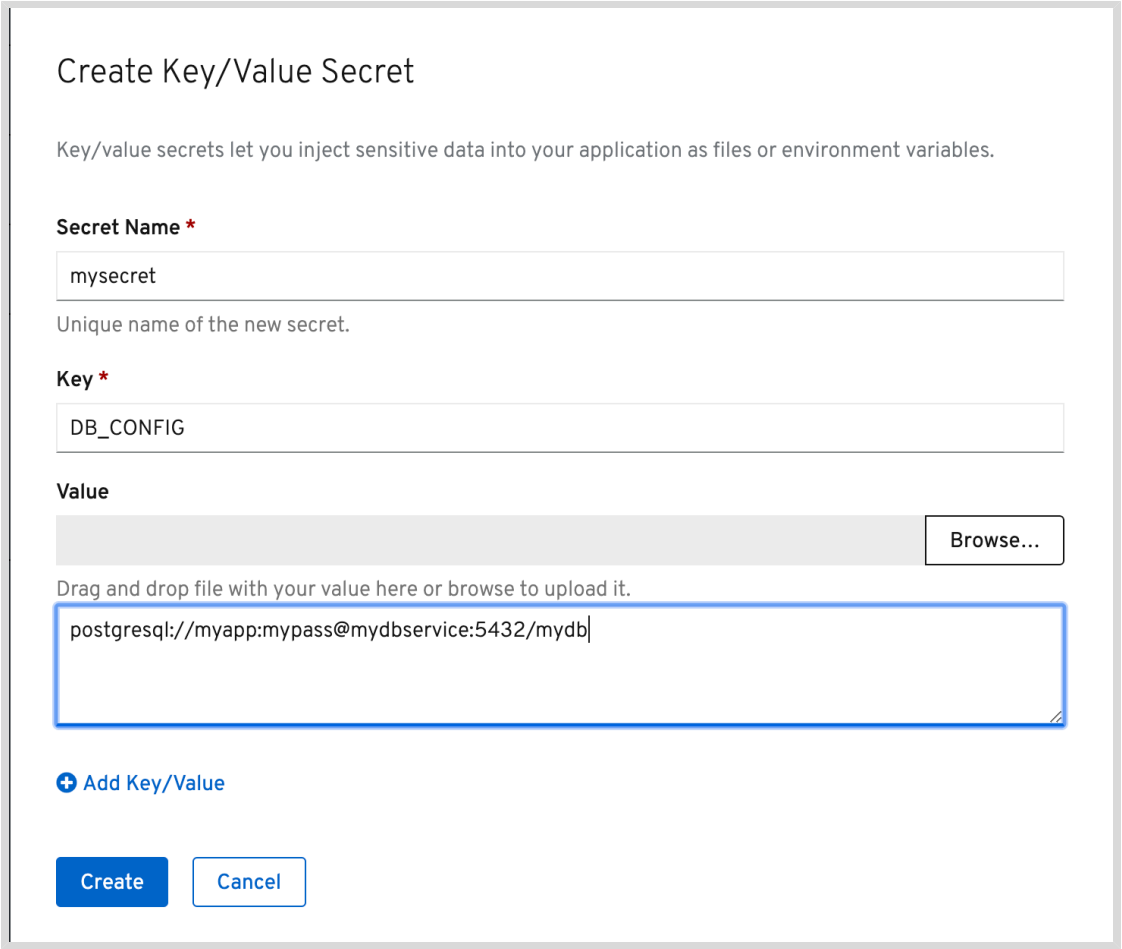


Figure 3.34: Secret details

4. After creating the secret, edit the deployment configuration for the application and map the secret to an environment variable accessible from the application.

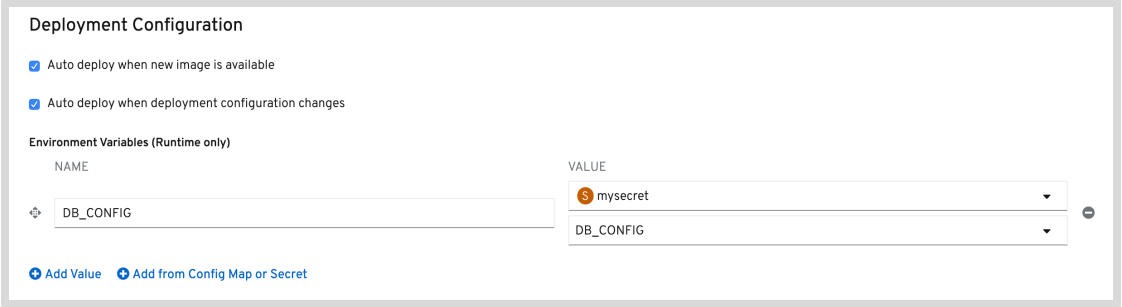


Figure 3.35: Map secret to environment variable

5. Finally, access the environment variable from a Node.js application as follows:

```
...output omitted...
const DB_CONFIG = process.env.DB_CONFIG ...output omitted...
const { Pool } = require('pg');

const pgconn = new Pool({
    connectionString: DB_CONFIG,
    ssl: false,
});
...output omitted...
```

References

For more information, refer to the Creating an application with a database section in the OpenShift Container Platform CLI Tools guide at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/cli_tools/index#creating-an-application-with-a-database

OpenShift and Databases

https://blog.openshift.com/openshift-connecting-database-using-port-forwarding/

OpenShift Port Forwarding

https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/nodes/index#nodes-containers-port-forwarding

Guided Exercise

Connecting an Application to a Database

In this exercise, you will deploy a Node.js application that connects to a PostgreSQL database on OpenShift.

Outcomes

You should be able to:

- Store the database connection information in a secret.
- Integrate the Node.js application with the PostgreSQL database using OpenShift services.
- Populate and fetch data from the PostgreSQL database using the Node.js application.

Before You Begin

To perform this exercise, ensure you have access to:

- A running Red Hat OpenShift Container Platform (RHOCP) cluster. See *Guided Exercise: Deploying an Application to Red Hat OpenShift Container Platform* for more information about using the developer sandbox RHOCP cluster.
- The source code for the `contacts` application in the `DO101x-apps` Git repository on your local system.
- The `oc` command.

Instructions

1. Inspect the source code for the `contacts` application.

1.1. Launch the Visual Studio Code (VS Code) editor, and then open the `DO101x-apps` folder in the `My Projects` workspace. The source code for the `contacts` application is in the `contacts` directory.

1.2. Inspect the `DO101x-apps/contacts/package.json` file to view the package dependencies for this Node.js application. The `contacts` application uses the popular *Express.js* web application framework, and it stores and fetches contact information from a PostgreSQL database.

```
...output omitted...
"dependencies": {
  "connect-flash": "0.1.1",
  "cookie-parser": "1.4.5",
  "debug": "4.3.2",
  "dotenv": "10.0.0",
  "express": "4.17.1",
  "http-errors": "1.8.0",
  "morgan": "1.10.0",
  "pg": "8.6.0",
```

```
  "pug": "3.0.2"
},
...output omitted...
```

1.3. Inspect the `DO101x-apps/contacts/app.js` file, which is the main entry point for the application. There is a single Express.js route definition called `indexRouter`:

```
app.use('/', indexRouter);
```

1.4. The code for the `indexRouter` route is defined in the `DO101x-apps/contacts/routes/index.js` file. Open this file in VS Code. The file contains the code to insert and fetch contact information from a PostgreSQL database.

1.5. The first method handles HTTP GET requests to the '/' URL. This method checks for the existence of the `contacts` table in the database. If the table does not exist, it renders an HTML page with a button to seed data in the database.

If the `contacts` table exists, the list of contacts is fetched as a JSON array and passed to the front-end view layer, which displays the contacts in an HTML table.

```
...output omitted...
router.get('/', function(req, res) {
...output omitted...
  res.render('index', { error: null, contacts: contacts, title: 'Contact List' });
});
...output omitted...
```

Note

The code for the HTML front-end is in the `DO101x-apps/contacts/views/index.pug` file.

1.6. The second method handles HTTP POST requests to the URL `/seed`. This method creates the `contacts` table, and populates it with a list of contacts. After the data is stored in PostgreSQL, the method redirects the request to the `/` URL, which renders the HTML table with the contacts.

```
...output omitted...
router.post('/seed', function(req,res) {
  pgconn.query("drop table if exists contacts; create table contacts ...output
 omitted...

    // redirect to the index page
    else {
      res.redirect('/');
    }
...output omitted...
```

1.7. The database connection information is configured in the `DO101x-apps/contacts/db/config.js` file. The database URL, consisting of the host name, port, database name, user name, and password is injected as an environment variable (`DB_CONFIG`) at run time. You will create an OpenShift secret to store the database URL:

```
...output omitted...
const DB_CONFIG = process.env.DB_CONFIG
...output omitted...
const { Pool } = require('pg');

const pgconn = new Pool({
    connectionString: DB_CONFIG,
    ssl: false,
});
...output omitted...
```

2. Create a new branch in your Git repository for the `contacts` application.

   2.1. In the source control view in VS Code (**View** > **SCM**), ensure that the `DO101x-apps` entry under `SOURCE CONTROL REPOSITORIES` shows the `main` branch. If you are working with another branch for a different exercise, click the current branch and then select `main` in the `Select a ref to checkout` window to switch to the `main` branch.

Warning

Each exercise uses a unique branch. Always create a new branch using `main` as the base.

   2.2. Click **main** in the `DO101x-apps` entry, for `SOURCE CONTROL REPOSITORIES`. Select `Create new branch...` from the list of options.

   2.3. At the prompt, enter `contacts` for the branch name. The Source Control view updates the `DO101x-apps` entry with the new branch name.

   2.4. Push the `contacts` branch to your `DO101x-apps` GitHub repository.

   Click the cloud icon for the `contacts` branch to push your local branch to your remote Git repository. When prompted, provide your GitHub user name and password.

3. Deploy a PostgreSQL database in the *`youruser`*`-dev` project.

   3.1. Log in to the OpenShift web console by using your developer account. Confirm the `Developer` perspective is active.

   3.2. Click **Add**, and then ensure that the current project is set to *`youruser`*`-dev`.

   3.3. Click **Database** in the **Developer Catalog** section.

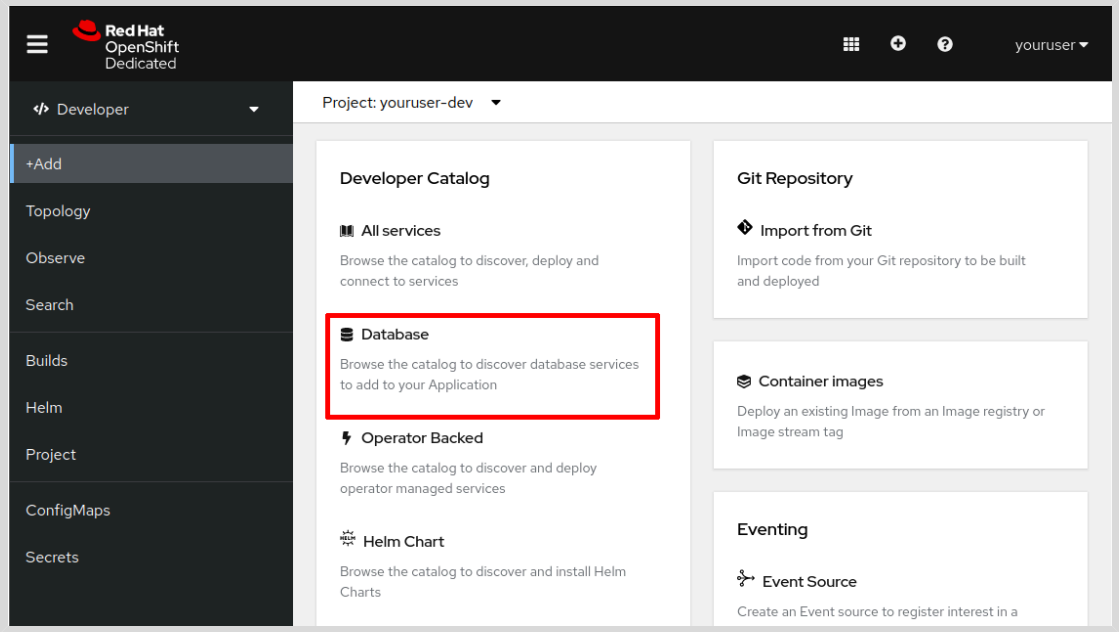


Figure 3.36: Add database

3.4. On the `Developer Catalog` page, click **Databases** > **Postgres**.

3.5. Click the `PostgreSQL (Ephemeral)` option.

Warning

If you do not see the `PostgreSQL (Ephemeral)` template, make sure the `Template` filter is selected in the **Type** section.

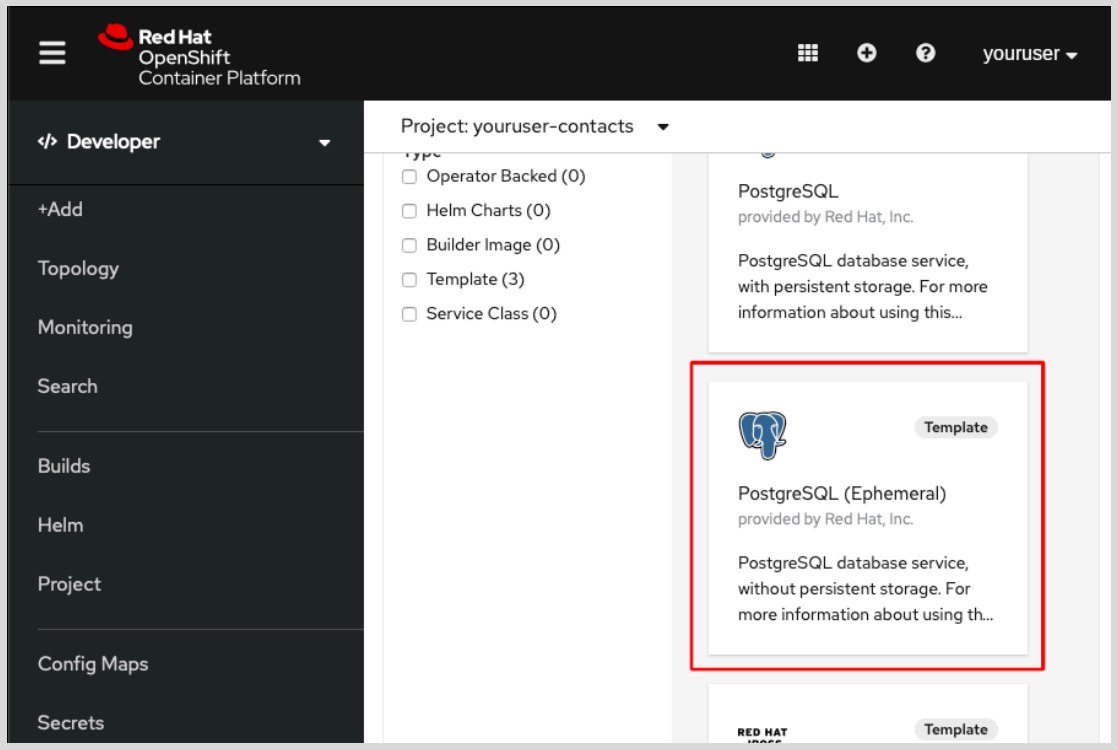


Figure 3.37: Add PostgreSQL database

3.6. Click **Instantiate Template**, and then complete the form according to the following table:

New Database Form

Form Field	Value
`Database Service Name`	`contactsdb`
`PostgreSQL Connection Username`	`contacts`
`PostgreSQL Connection Password`	`contacts`
`PostgreSQL Database Name`	`contactsdb`

Leave all other fields at their default values.

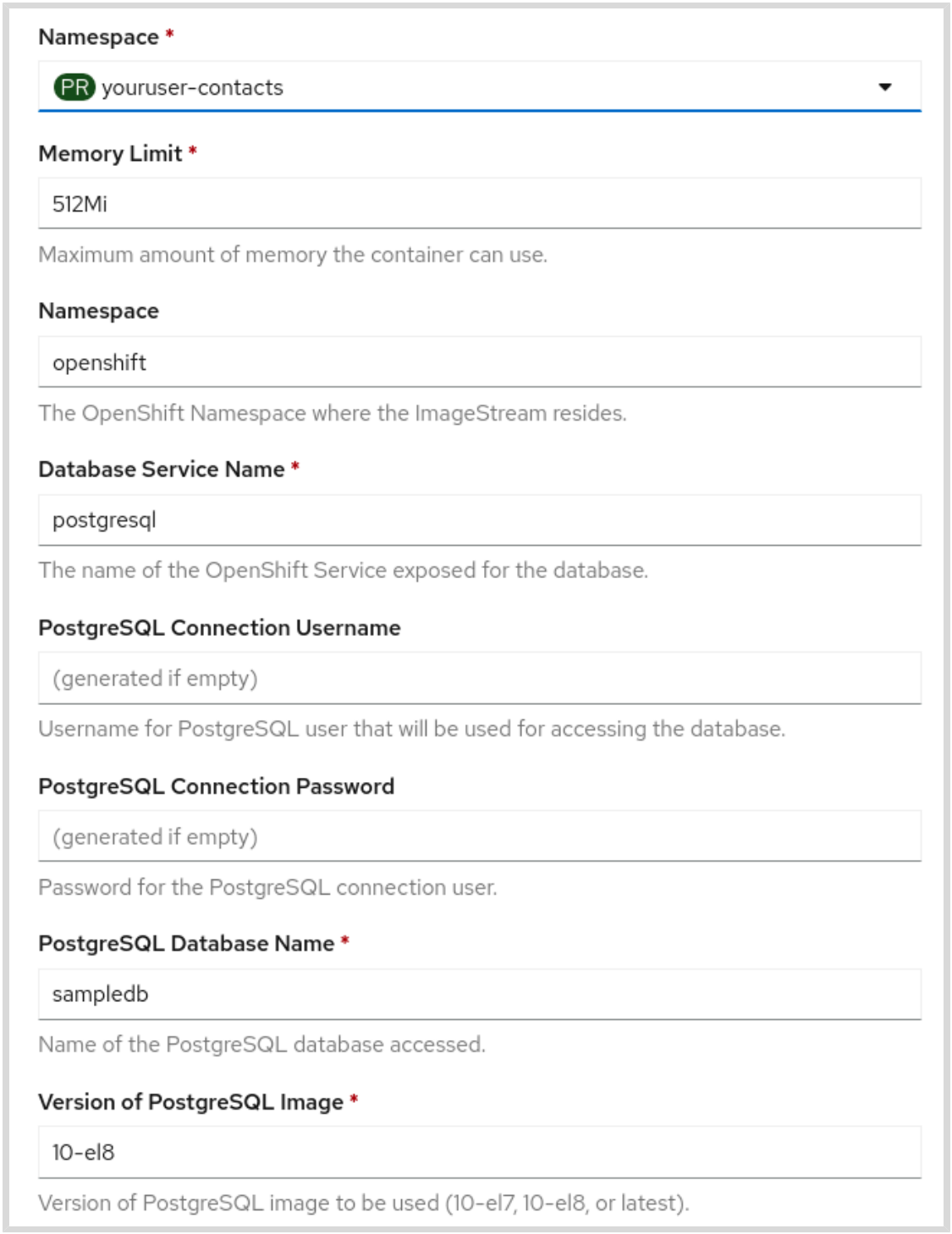


Figure 3.38: New PostgreSQL database form

Click **Create** to start the deployment of the database.

3.7. Click **Topology** in the left side menu, and then click the `contactsdb` deployment. Click `Resources`, and then verify that a single PostgreSQL database pod is running.

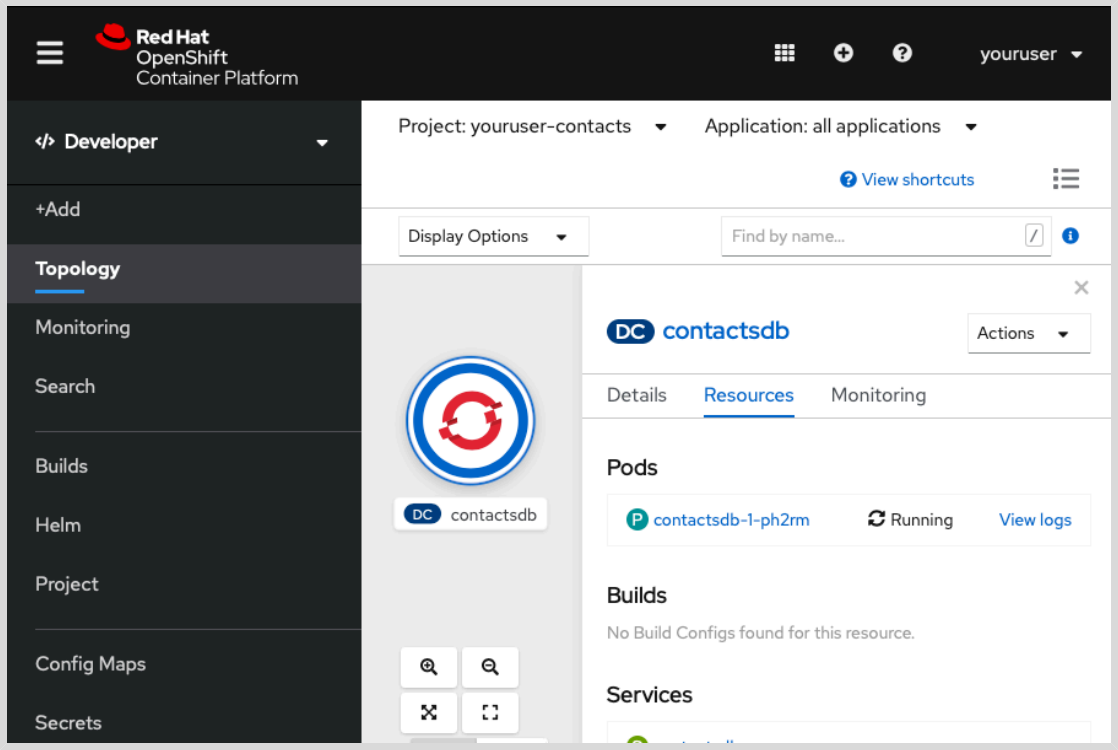


Figure 3.39: PostgreSQL database deployment

4. Create a secret to store the database connection information.

   4.1. Select `Secrets` in the left menu.

   4.2. Click **Create** > **Key/Value Secret** to create a new secret.

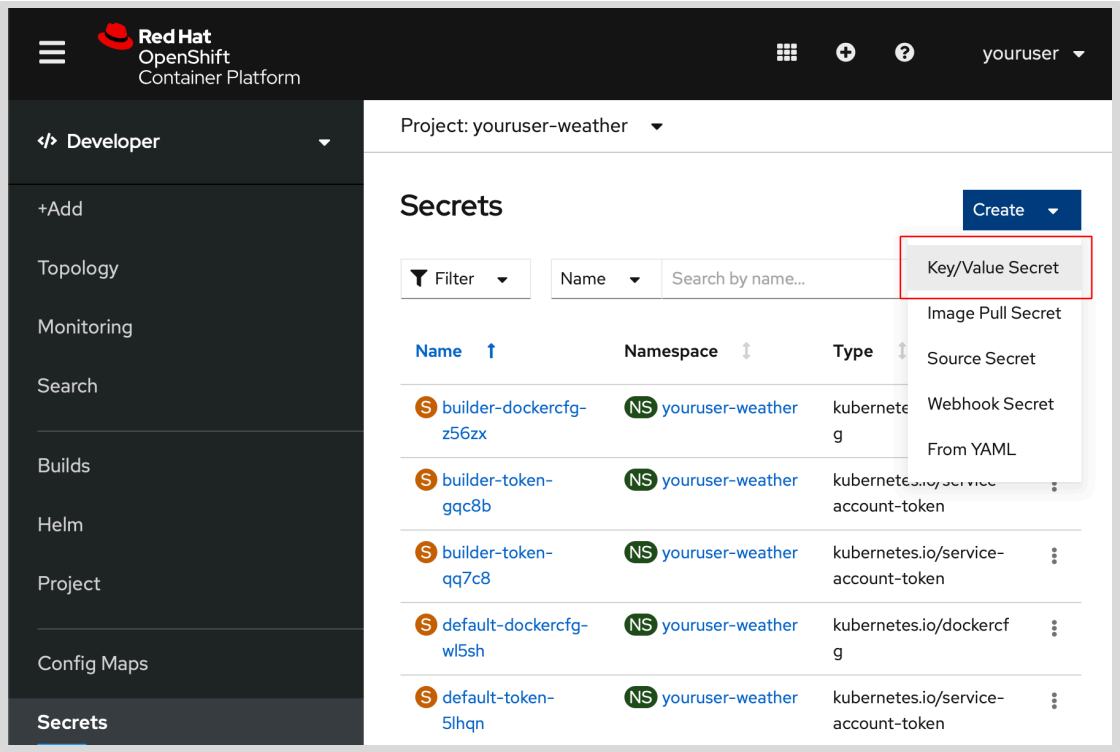


Figure 3.40: Create secret

4.3. On the `Create Key/Value Secret` page, enter `contactsdb-secret` in the `Secret Name` field, `DB_CONFIG` in the `Key` field, and `postgresql://contacts:contacts@contactsdb:5432/contactsdb` in the `Value` field.

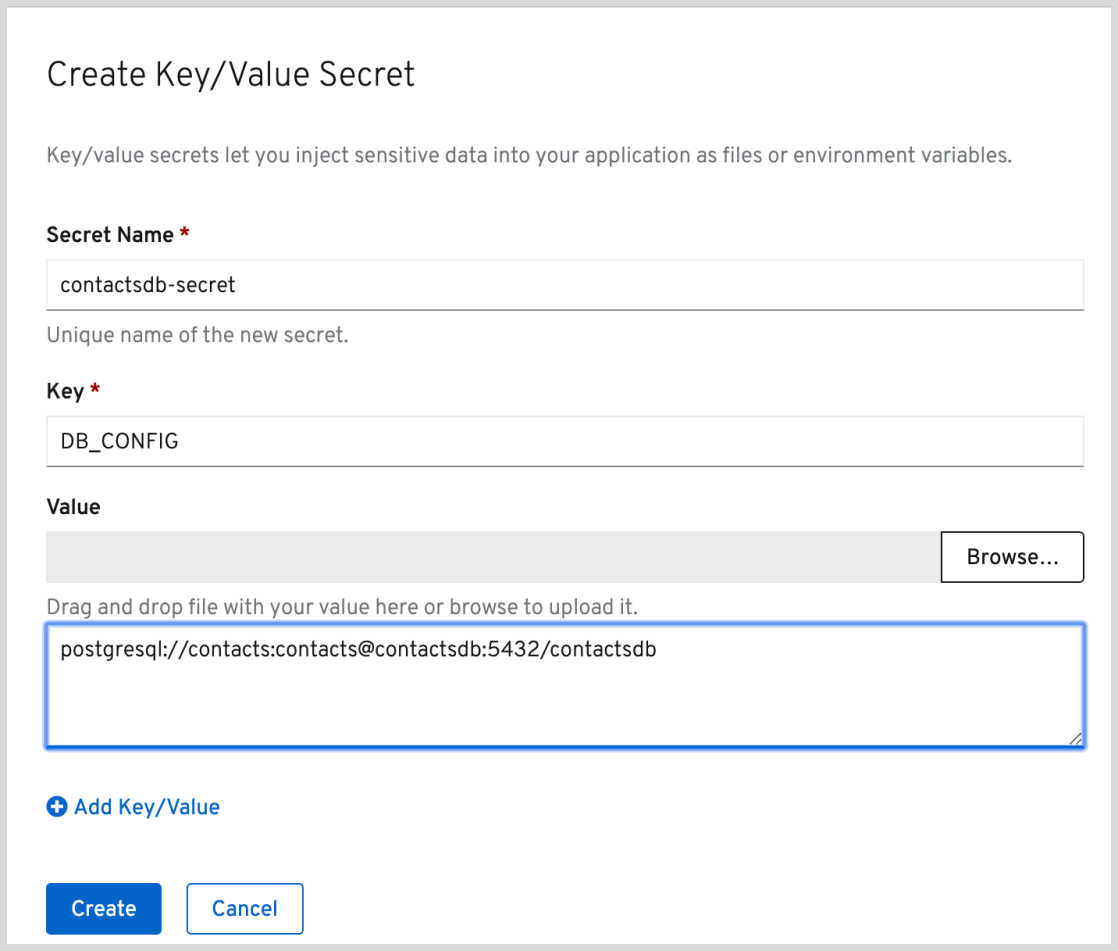


Figure 3.41: Secret details

Recall that the `DB_CONFIG` environment variable is mapped to the `connectionString` attribute in the `DO101x-apps/contacts/db/config.js` file. The PostgreSQL database driver module for Node.js expects the database URL to be of the form:

```
postgresql://<username>:<password>@<host>:<port>/<database>
```

4.4. Click **Create** to create the secret.

5. Deploy the `contacts` application to OpenShift.

5.1. Select `Add`, and then click **All services** in the **Developer Catalog** section.

5.2. Select **Languages** > **JavaScript**, and then click the `Node.js` builder image option.

> **Warning**
> If you do not see the `Node.js` builder image, make sure the `Builder Image` filter is selected in the **Type** section.

Click **Create Application** to enter the details of the application.

5.3. Complete the form according to the following table. To access the Git parameters, click **Show Advanced Git Options**.

New Application Parameters

Form Field	Value
`Git Repo URL`	`https://github.com/yourgituser/DO101x-apps`
`Git Reference`	`contacts`
`Context Dir`	`/contacts`
`Application Name`	`contacts`
`Name`	`contacts`

Do not click **Create** yet. First, you must customize the deployment for the application to make use of the secret that you created in a previous step.

5.4. Click `Deployment` to customize the deployment options. Reference the secret that you created earlier.

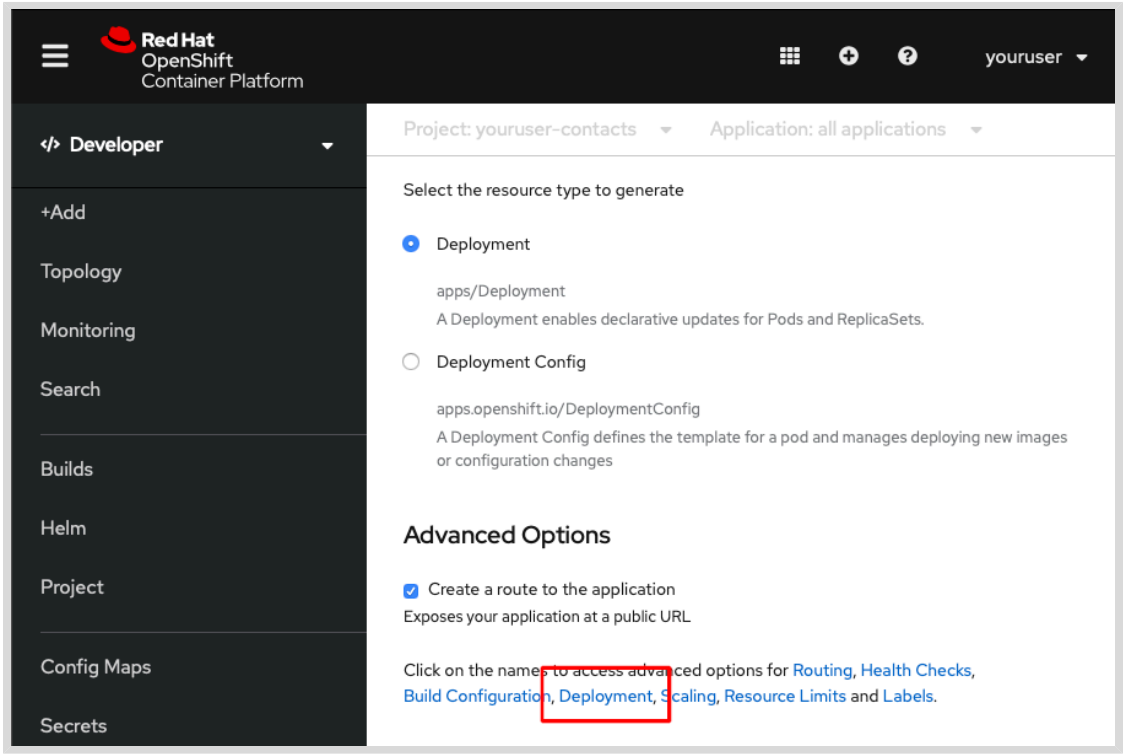


Figure 3.42: Deployment for contacts application

Click **Add from Config Map or Secret**

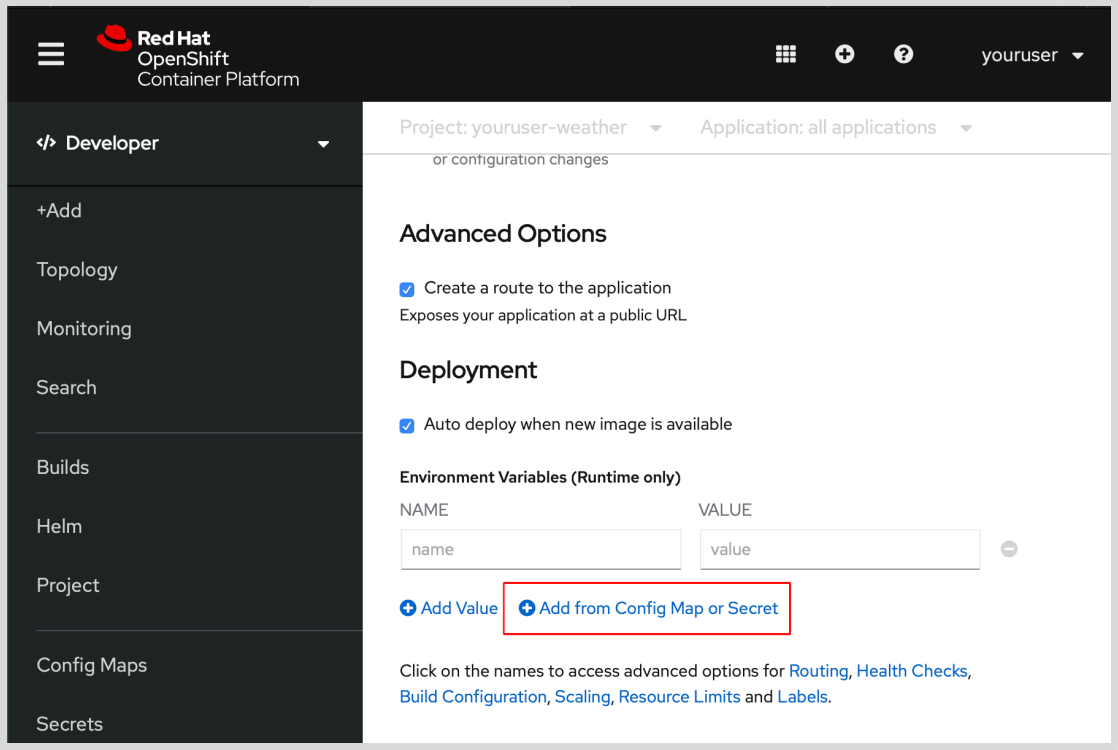


Figure 3.43: Add environment variables from config map or secret

5.5. A new row is added to the `Environment Variables` table. Enter `DB_CONFIG` in the `NAME` field in the new row. Click the `Select a resource` list and select `contactsdb-secret`.

5.6. Click the `Select a key` list and select `DB_CONFIG`.

5.7. Remove the first row in the `Environment Variables` table by clicking the minus (-) icon next to the empty `value` field.

Your final `Environment Variables` table should display as follows:

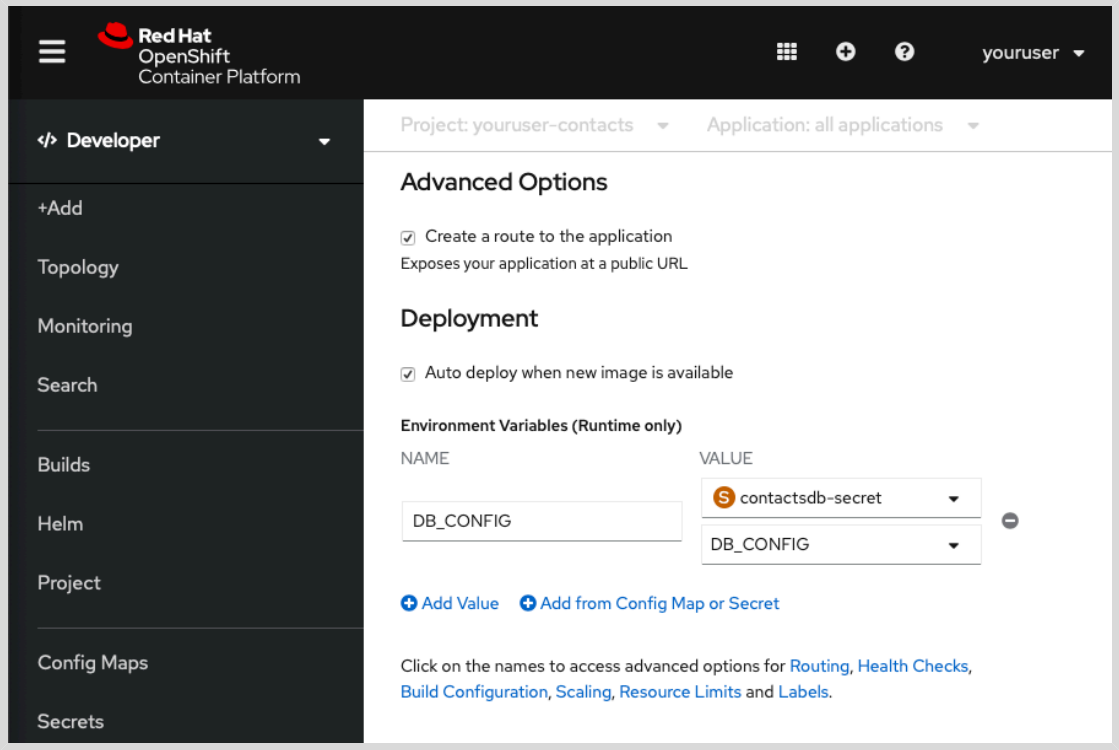


Figure 3.44: Final environment variables table

5.8. To avoid deployment errors in the following steps, review the values you entered in the form before proceeding. Click **Create** to start the build and deployment processes.

6. Test the application.

6.1. Wait for a few minutes while the application container image is built and deployed. You should see a green tick mark for the `contacts` deployment on the `Topology` page.

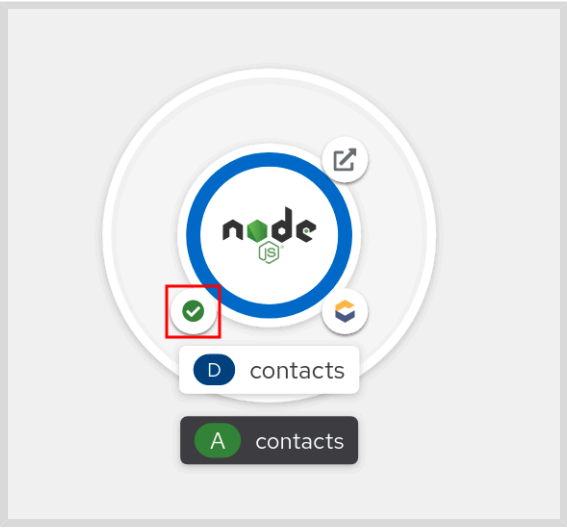


Figure 3.45: Contacts app built successfully

6.2. Click `Open URL` to open the route URL for the `contacts` application.

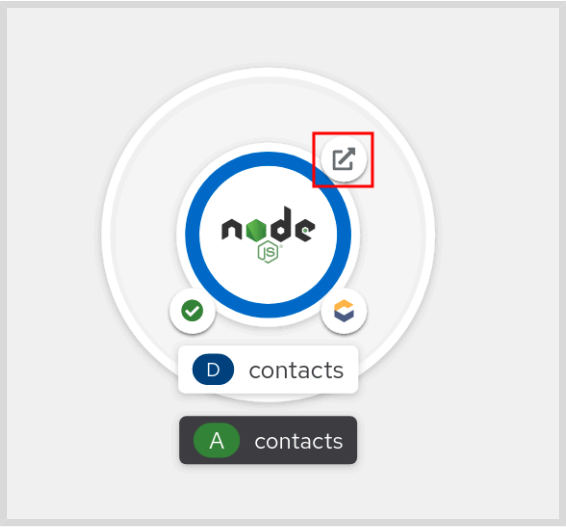


Figure 3.46: Contacts app route URL

6.3. The home page for the `contacts` application displays.

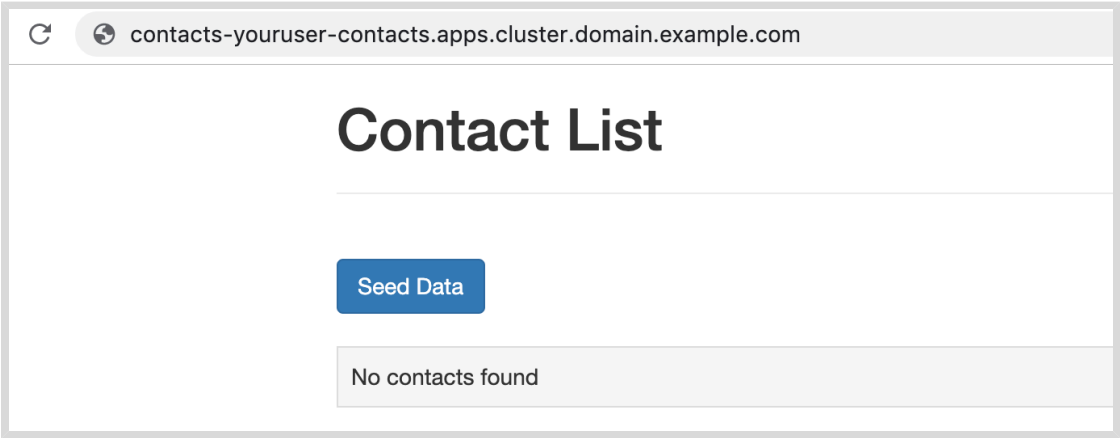
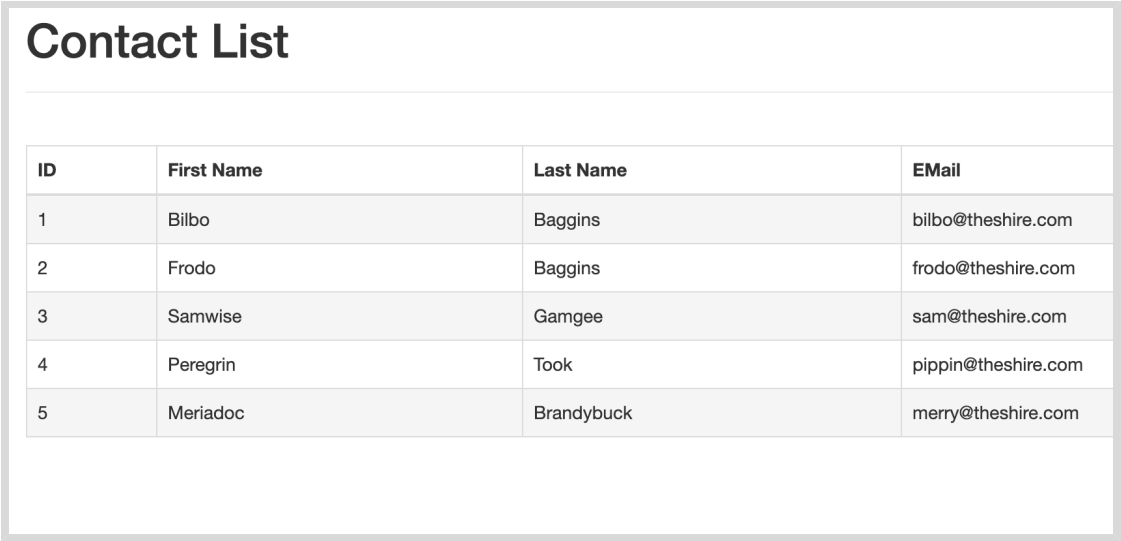


Figure 3.47: Contacts app home page

6.4. Click **Seed Data** to populate the database with sample contacts.

6.5. A table with five sample contacts that are fetched from the database displays.



Contact List

ID	First Name	Last Name	EMail
1	Bilbo	Baggins	bilbo@theshire.com
2	Frodo	Baggins	frodo@theshire.com
3	Samwise	Gamgee	sam@theshire.com
4	Peregrin	Took	pippin@theshire.com
5	Meriadoc	Brandybuck	merry@theshire.com

Figure 3.48: Sample contacts

Close the browser tab.

7. Verify contacts data in the PostgreSQL database using the OpenShift web console.

7.1. Click **Topology** in the left side menu.

7.2. Click on the `contactsdb` deployment, and then click `Resources`.

7.3. Click the `contactsdb` pod, listed in the `Pods` section. The pod name will differ from this example.

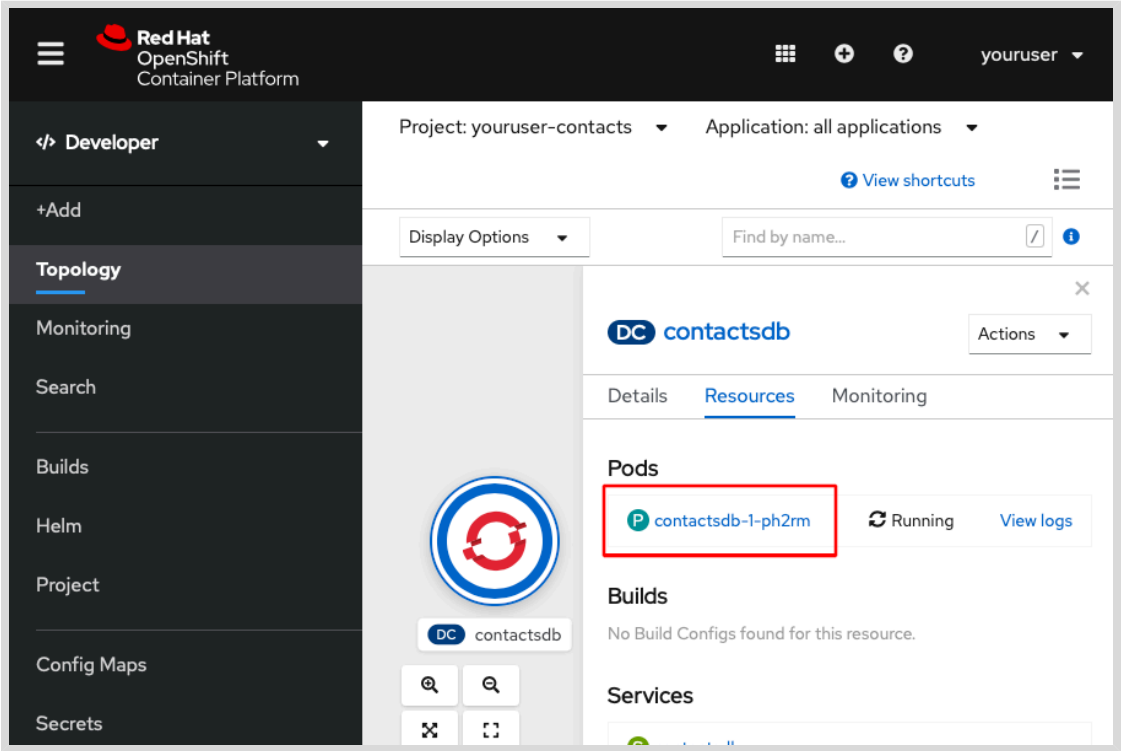


Figure 3.49: Click contactsdb pod

7.4. On the pod details page, click `Terminal` to open a command line terminal session inside the running PostgreSQL database pod.

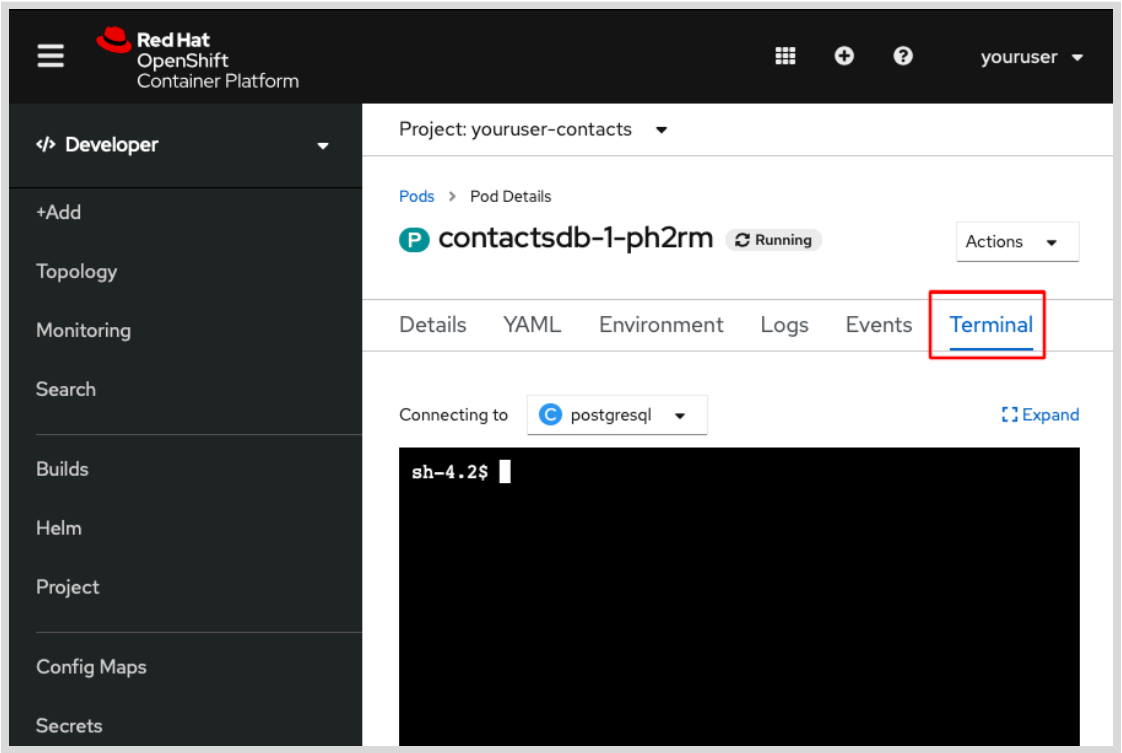


Figure 3.50: Click terminal tab

7.5. The PostgreSQL database client program `psql` is available to access the database. Run the following commands in the terminal to verify that the `contacts` table exists in the `contactsdb` database, and has five rows in it.

First, connect to the `contactsdb` database.

```
sh-4.2$ psql -U contacts contactsdb
```

Next, list the tables in the database and verify that a table called `contacts` is displayed.

```
contactsdb=> \dt
```

Finally, run a select query to list the data in the `contacts` table.

```
contactsdb=> select * from contacts;
```

The following output displays.

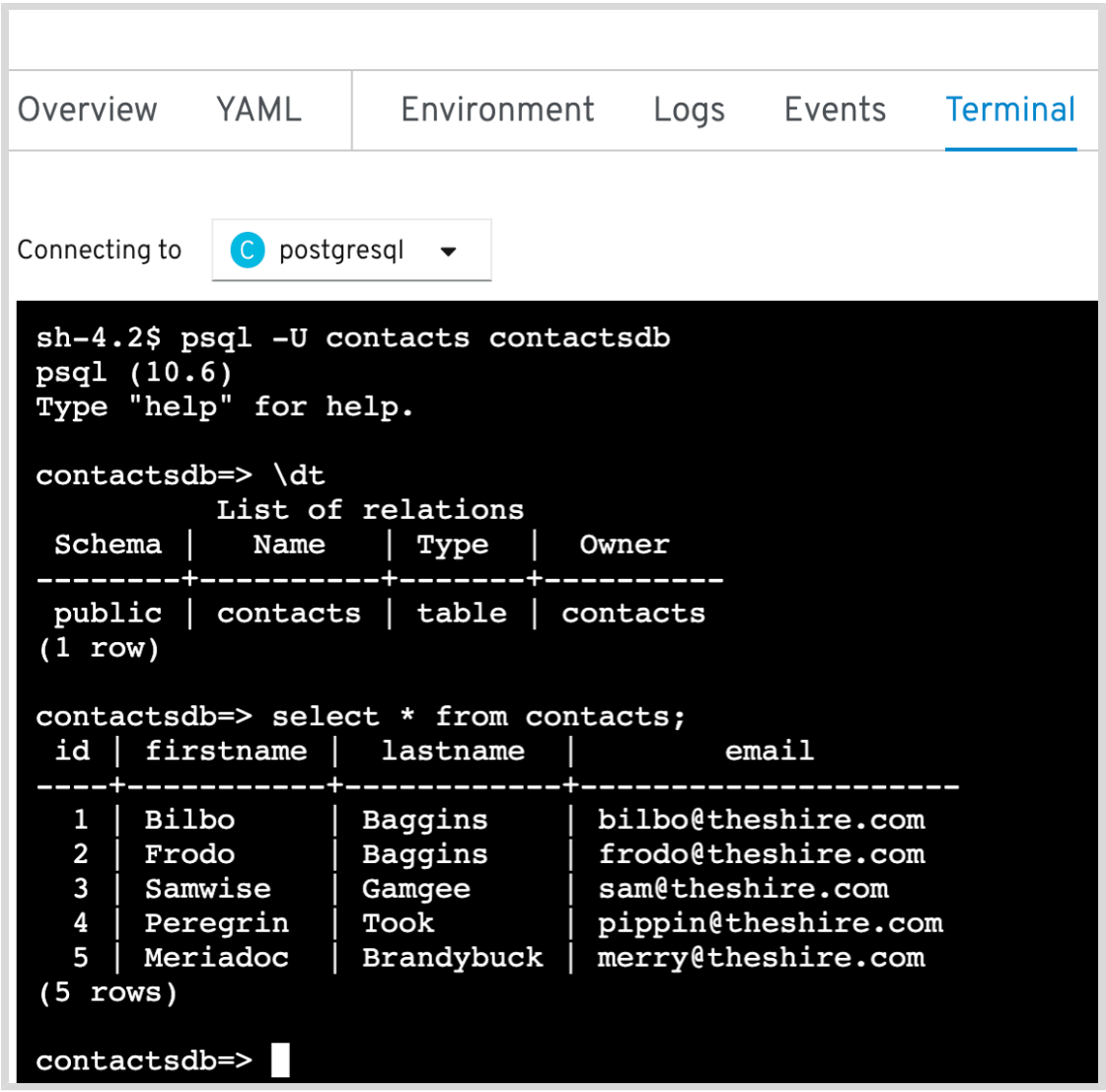


Figure 3.51: Database output

8. Clean up.

   8.1. In the RHOCP web console, click your username at the top right, and then click **Copy Login Command**. This will open a new tab and prompt you for your username and password.

   8.2. Once authenticated, click **Display Token** and copy the provided `oc login` command.

   8.3. Log in to your RHOCP account by using the copied command and notice the default project:

```
$ oc login --token=yourtoken --server=https://api.region.prod.nextcle.com:6443
Logged into "https://api.region.prod.nextcle.com:6443" as "youruser" using the
 token provided.
...output omitted...
Using project "youruser-dev".
```

   8.4. Delete RHOCP objects with the `app=contacts` label:

```
$ oc delete all --selector app=contacts
pod "contacts-5bdc7dbfb4-hdq48" deleted
service "contacts" deleted
deployment.apps "contacts" deleted
buildconfig.build.openshift.io "contacts" deleted
imagestream.image.openshift.io "contacts" deleted
route.route.openshift.io "contacts" deleted
```

8.5. Delete RHOCP objects with the `template=postgresql-ephemeral-template` label:

```
$ oc delete all,secret --selector template=postgresql-ephemeral-template
replicationcontroller "contactsdb-1" deleted
service "contactsdb" deleted
deploymentconfig.apps.openshift.io "contactsdb" deleted
secret "contactsdb" deleted
```

8.6. Delete the `contactsdb-secret`, `contacts-github-webhook-secret`, and `contacts-generic-webhook-secret` secrets:

```
$ oc delete secret contactsdb-secret contacts-generic-webhook-secret
 contacts-github-webhook-secret
secret "contactsdb-secret" deleted
secret "contacts-generic-webhook-secret" deleted
secret "contacts-github-webhook-secret" deleted
```

Finish

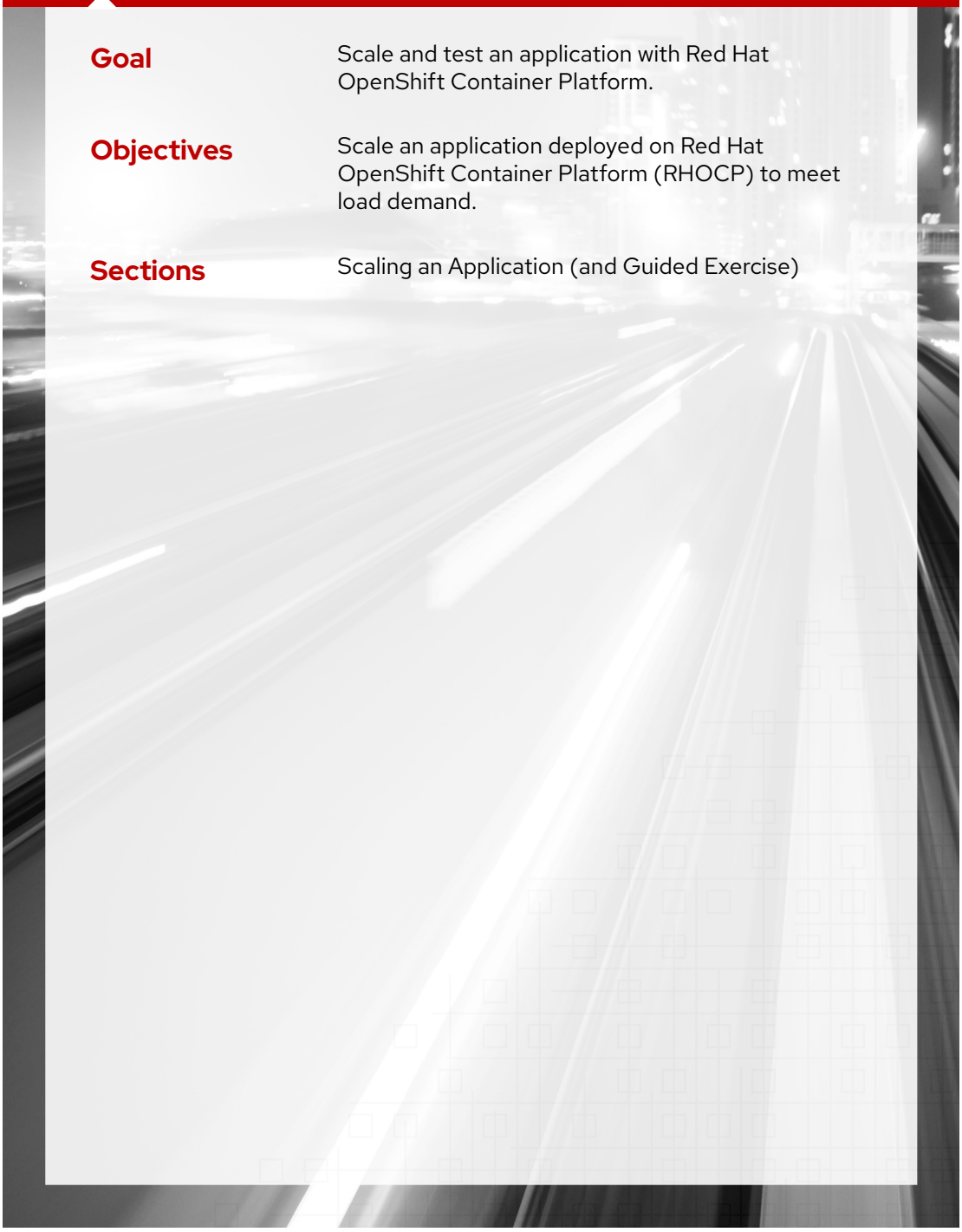
This concludes the guided exercise.

Summary

- You can manually trigger new builds using the OpenShift web console or the OpenShift CLI.
- You can automatically trigger new builds using Webhooks.
- You can externalize the application configuration using secrets and configuration maps.
- Secrets are used to store sensitive information like passwords.
- Configuration maps are used to store non-sensitive plain text configuration parameters.
- Applications can store database connection credentials and other information in a secret, and integrate with databases using OpenShift services.

Chapter 4

Scaling Applications in OpenShift



Goal	Scale and test an application with Red Hat OpenShift Container Platform.
Objectives	Scale an application deployed on Red Hat OpenShift Container Platform (RHOCP) to meet load demand.
Sections	Scaling an Application (and Guided Exercise)

Scaling an Application

Objectives

After completing this section, you should be able to scale an application deployed on Red Hat OpenShift Container Platform (RHOCP) to meet load demand.

Describing Pod Scaling

Most real-world applications do not run only in a single pod. They often need to run in several pods to meet growing user demand. By duplicating the application in multiple pods, the application is **scaling** to meet user demand. When a user makes a request to access the application, RHOCP automatically directs the service request to an available pod. If more pods running the application are available, then users are less likely to experience an outage or unavailable application.

Some applications receive a large number of concurrent requests only during certain periods. This makes it difficult to determine the number of needed pods before running the application. However, there are extra costs associated with running more pods than required when traffic is not at its peak.

RHOCP refers to the action of changing the number of pods for an application as **scaling**. **Scaling up** increases the number of pods for an application. **Scaling down** decreases the number of pods. Scaling up enables the application to handle more client requests, and scaling down provides cost savings as the load drops.

For users to reach an application from outside RHOCP, you must create a route resource that associates a public URL to the application. When scaling your application, RHOCP automatically configures that route to distribute client requests among member pods.

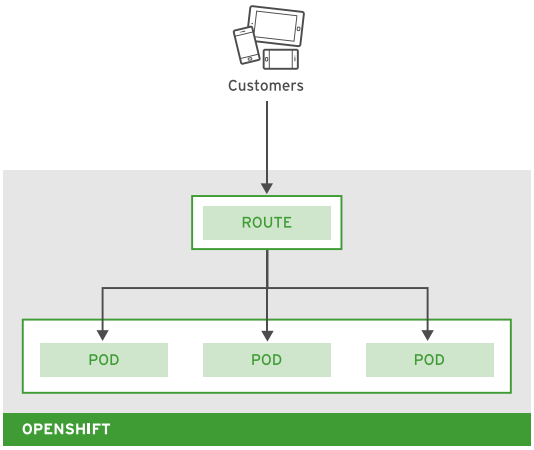


Figure 4.1: The route resource distributes client requests

When scaling up an application, RHOCP first deploys a new pod and then waits for the pod to be ready. Only after the new pod becomes available does RHOCP configure the route to send traffic to the new pod. When scaling down, RHOCP reconfigures the route to stop sending traffic to the pod, and then deletes the pod.

Preparing your Application for Scaling

RHOCP allows any application to scale by creating multiple pods, but this does not mean that every application automatically becomes scalable because it is running in RHOCP. The application must be able to work correctly with multiple instances of itself.

For example, some web applications maintain user state, such as a cookie-based HTTP session. Because a user can be directed to any of the running pods, that session data must be available to all of the pods and not just the first one the user reaches. These applications might keep the session data in a central store, such as a database or in-memory shared store. They cannot only store the data in the pod local file system.

Additionally, database servers, such as MariaDB and PostgreSQL, often do not support running in multiple pods by default. One way to scale database servers is to use Operators.

Configuring the Route Resource

By default, RHOCP tries to direct all requests from a client to the same pod. You can configure the route resource to modify this behavior.

If the round-robin algorithm is selected, RHOCP distributes the requests in a round-robin fashion between the pods. In the case of two pods, RHOCP sends the first request to the first pod, the second request to the second pod, the third request to the first pod, and so on.

Note
Distributing requests between multiple servers is also called *load balancing*.

To edit the route details:

1. Navigate to the **Topology** page.
2. Click the application icon.
3. Click the **Overview** tab.
4. Select the route resource.
5. Modify the `annotations` section in the YAML file to edit the route parameters.

Scaling an Application Manually

With the RHOCP web console, developers can manually scale their applications. For example, the developer may increase the number of pods when anticipating a surge in the application load.

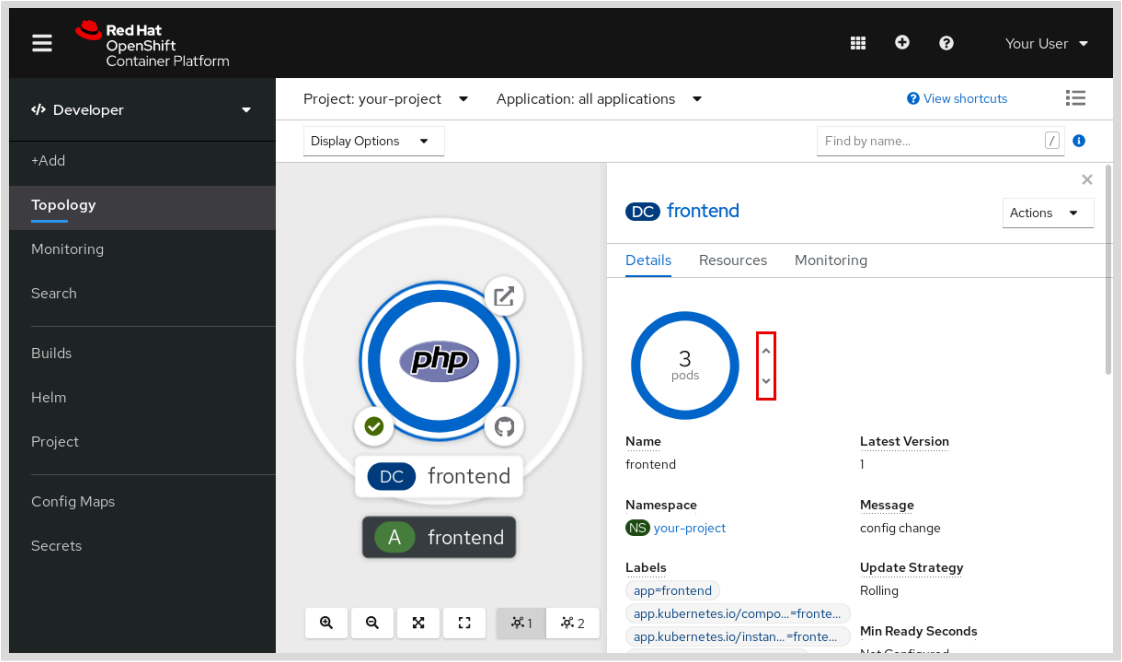


Figure 4.2: Scaling an application with the web console

Configuring the Horizontal Pod Autoscaler

In addition to manual scaling, RHOCP provides the `Horizontal Pod Autoscaler (HPA)` feature. An HPA automatically increases or decreases the number of pods depending on average CPU utilization. Developers can also configure an HPA to use custom application metrics for scaling. Although this advanced configuration is outside the scope of this course, the reference section provides a link for more information.

In the following example, the command enables and configures an HPA for the `frontend` deployment:

```
[student@workstation ~]$ oc autoscale deployment/frontend --min=1 --max=5
 --cpu-percent=80
```

The options are as follows:

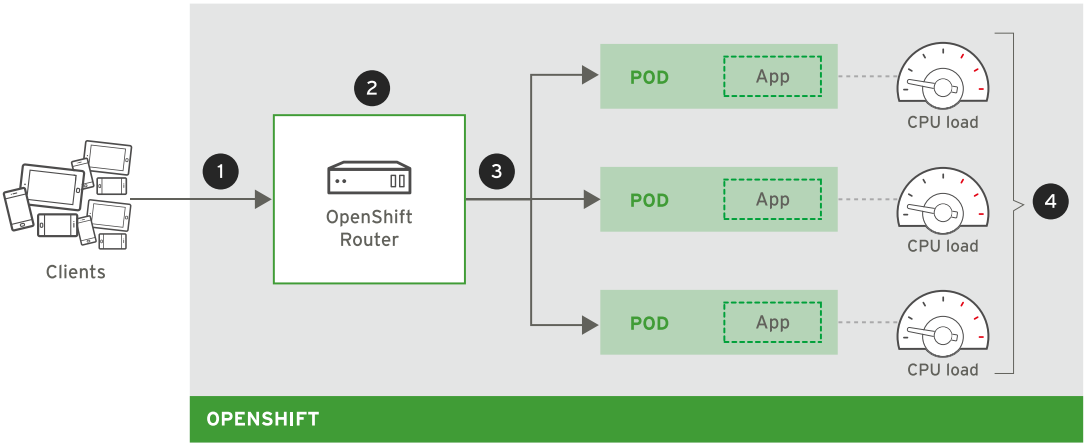
deployment/frontend
Name of the application deployment.

--min=1
Minimum number of pods.

--max=5
Maximum number of pods.

--cpu-percent=80
Ideal average CPU utilization for each pod. If the global average CPU utilization is above that value, then HPA starts a new pod. If the global average CPU utilization is below the value, then HPA deletes a pod.

The following diagram shows how HPA scales an application based on CPU utilization.



1. Clients access the application.
2. The RHOCP router distributes the requests to the pods.
3. The application in the pods processes the request.
4. When the number of clients increases, the CPU utilization increases in each pod. When the CPU utilization is above the configured value, HPA scales up by deploying a new pod.

References

For more information, refer to the Scaling application pods and checking builds and routes section in the `Creating and managing applications on RHOCP` guide at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/applications/index#odc-scaling-application-pods-and-checking-builds-and-routes_viewing-application-composition-using-topology-view

For more information on scaling pods, refer to the `Automatically scaling pods` section in the `Configuring and managing nodes in RHOCP` guide at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/nodes/index#nodes-pods-autoscaling

For more information on using custom metrics with HPA, refer to the `Exposing custom application metrics for autoscaling` section in the `Configuring and using the monitoring stack in RHOCP` guide at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/monitoring/index#exposing-custom-application-metrics-for-autoscaling

Guided Exercise

Scaling an Application

In this exercise, you will configure Red Hat OpenShift Container Platform (RHOCP) to automatically scale up an application to meet load demand.

Outcomes

You should be able to use RHOCP to:

- Manually scale an application using the web console.
- Configure RHOCP to automatically scale an application when the CPU load increases.

Before You Begin

To perform this exercise, ensure that you have access to the following:

- An RHOCP cluster. See *Guided Exercise: Deploying an Application to Red Hat OpenShift Container Platform* for more information about using the developer sandbox RHOCP cluster.
- Visual Studio Code (VS Code).
- Git.
- The `oc` command.

Additionally, you should have your fork of the `DO101x-apps` repository cloned and opened in VS Code.

Note

Install the `oc` command-line utility by following the instructions in *Guided Exercise: Updating an Application* .

Instructions

1. Using VS Code, create a new branch named `scale` in the `DO101x-apps` repository. Push the branch to your GitHub repository.

   Each exercise uses a unique branch. Always create a new branch using `main` as the base.

   1.1. Open VS Code. Click **View** > **SCM** to access the source control view. Ensure that the `DO101x-apps` entry under `SOURCE CONTROL PROVIDERS` shows the `main` branch.

   If you were working with another branch for a different exercise, click the current branch and then select `main` in the `Select a ref to checkout` window.

Note

If the Source Control view does not display the `SOURCE CONTROL PROVIDERS` heading, then right-click `SOURCE CONTROL` at the top of the Source Control view and select `Source Control Providers`.

1.2. Click **main** in the `DO101x-apps` entry under `SOURCE CONTROL PROVIDERS`, and then select `Create new branch`. Type `scale` to name the branch.

1.3. Click the **Views and more actions** > **push** to publish the changes to the remote repository.

If prompted that this action will push and pull commits to and from the origin, then click **OK** to continue. If prompted, to authorize VS Code to push commits, then click **Allow** and continue on the GitHub website.

Alternatively, you can choose not to authorize GitHub. In such case, you will be prompted for username and password. Enter your GitHub developer token as your username, and leave the password field empty.

2. Deploy the `scale` PHP application. When accessed from the internet, this application displays the name and IP address of its pod.

The source code for this application is in the `php-scale` subdirectory of your Git repository. Use the `scale` branch to deploy the application to RHOCP.

2.1. Access the RHOCP web console and log in. Confirm the `Developer` perspective is active.

2.2. Click **Add**, and then ensure that the current project is set to `youruser-dev`.

2.3. Click **All services** in the **Developer Catalog** section.

2.4. Click **Languages** > **PHP**, and then **Type** > **Builder Image**. Make sure no other filtering options are selected. Click the PHP builder image to select it.

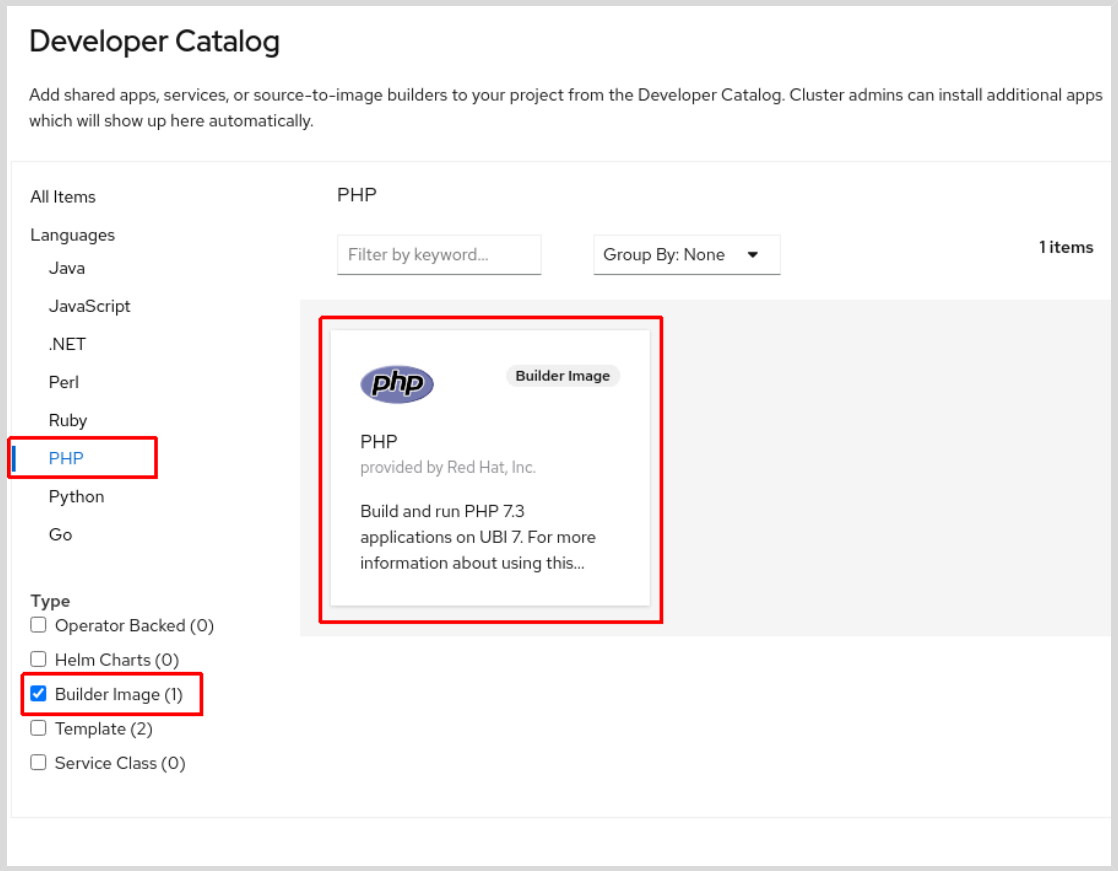


Figure 4.3: Creating a PHP application

Click **Create Application** to enter the details of the application.

2.5. Complete the form according to the following table. To access the Git parameters, click **Show Advanced Git Options**.

New Application Parameters

Parameter	Value
Git Repo URL	`https://github.com/yourgituser/DO101x-apps`
Git Reference	`scale`
Context Dir	`/php-scale`
Application Name	`scale`
Name	`scale`

Select `Deployment` as the resource type to generate.

To avoid unexpected errors, review the values that you entered in the form before proceeding. Click **Create** to start the build and deployment processes.

3. Wait for RHOCP to build and deploy your application and verify that you can access it from the internet.

3.1. The web console should automatically show the `Topology` page. Otherwise, manually click the **Topology** tab.

3.2. Wait for the application icon to show that the build and deployment are finished.

Figure 4.4: The scale application finished deploying

Once they have finished, click the **Open URL** icon to access the application.

Figure 4.5: Accessing the scale application

A new browser tab displays the application.

Figure 4.6: Displaying the scale application

Note

The host name and IP address displayed in the application will likely differ on your system.

3.3. Close the application browser tab.

4. Scale the application to two pods and confirm that you can still access it.

4.1. From the `Topology` page, click the PHP icon, and then click the **Details** tab.

4.2. Click the up arrow to scale the application to two pods.

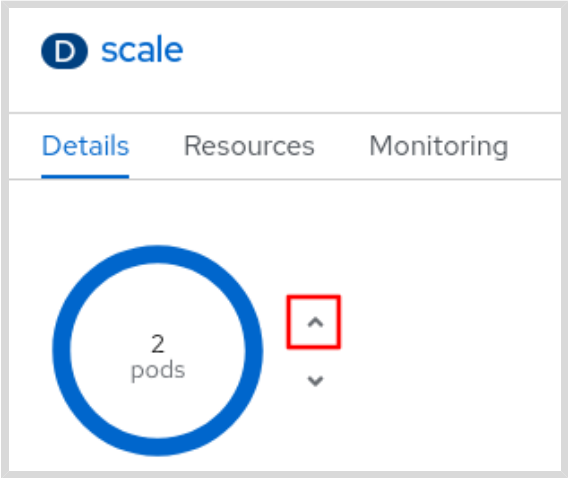


Figure 4.7: Scaling up the application

4.3. Wait a few seconds for the second pod to start, and then click the **Open URL** icon to confirm that you can still access the application.

4.4. Close the application browser tab.

5. By default, the load balancer redirects all requests from a particular client to the same pod.

Configure the route resource to disable the affinity between clients and pods. Then, use your web browser to test that RHOCP distributes requests between the two pods.

5.1. From the `Topology` page, click the PHP icon and then the **Resources** tab.

5.2. Click the **scale** route resource.

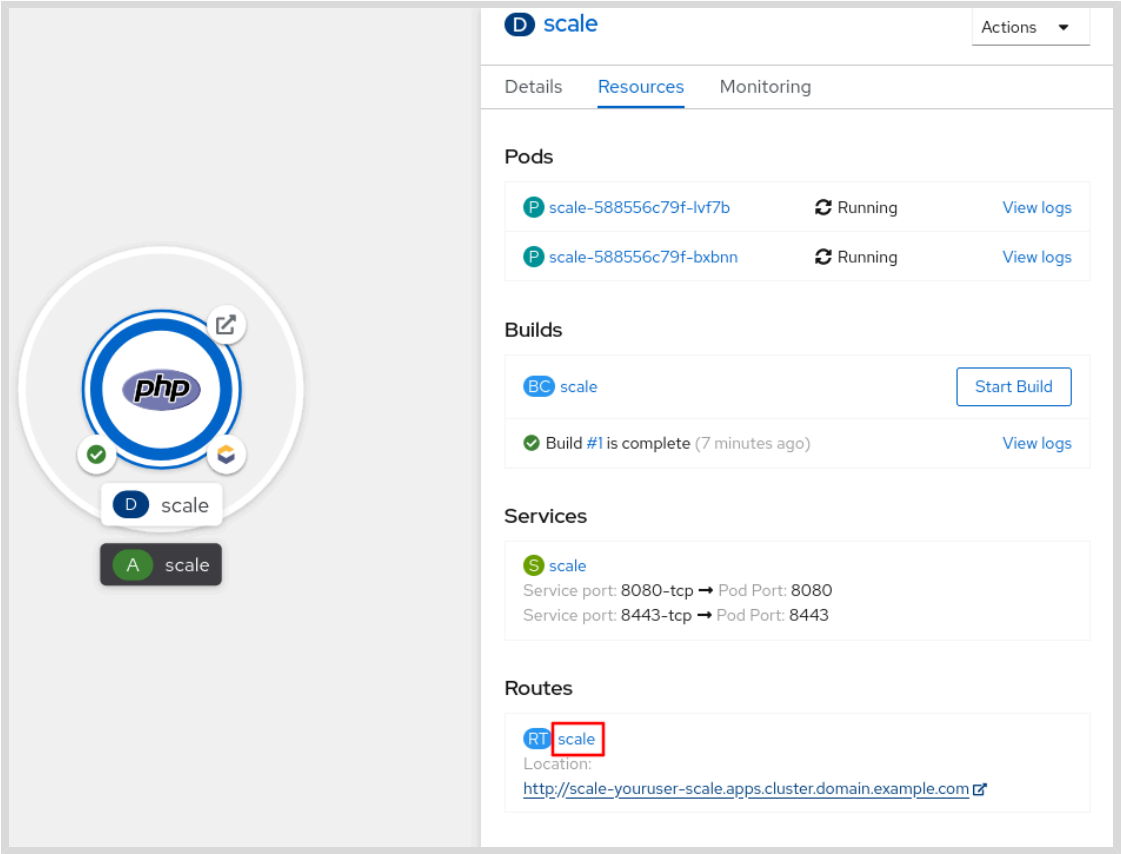


Figure 4.8: Accessing the route resource

5.3. Click the **YAML** tab, and then use the editor to add the following two lines in the `annotations` section:

```
annotations:
  openshift.io/host.generated: 'true'
  haproxy.router.openshift.io/balance: roundrobin
  haproxy.router.openshift.io/disable_cookies: 'true'
```

Important

Ensure that you maintain correct indentation as shown in the example. YAML files are indentation sensitive.

Save the changes to reconfigure the load balancer.

5.4. Use your web browser to confirm that RHOCP distributes requests between the two pods.

Open the application again and refresh the page several times. Notice that the pod name and IP address alternate with every request.

5.5. Close the application browser tab.

6. Scale down the application to one pod.

6.1. From the `Topology` page, click the PHP icon, and then click the **Details** tab.

6.2. Click the down arrow to scale the application to one pod.

6.3. Wait for the second pod to terminate, and then click the PHP application **Open URL** button to access the application again.

Refresh the application page several times. Notice that the page always displays the same pod name and IP address. RHOCP directs all requests to the remaining pod.

6.4. Close the application browser tab.

7. Configure RHOCP to automatically scale up your application when the CPU load is above 80%.

7.1. In the RHOCP web console, click your username at the top right, and then click **Copy Login Command**. This will open a new tab and prompt you for your username and password.

7.2. Once authenticated, click **Display Token** and copy the provided `oc login` command.

Important

Because this token is used for authentication, treat it as your RHOCP password.

7.3. Log in to your RHOCP account by using the copied command:

```
$ oc login --token=yourtoken --server=https://api.region.prod.nextcle.com:6443
Logged into "https://api.region.prod.nextcle.com:6443" as "youruser" using the
 token provided.
...output omitted...
Now using project "youruser-dev" on server ...output omitted...
```

7.4. Configure the autoscaler for the application. Set the maximum number of pods to three and the CPU load to 80%.

```
$ oc autoscale deployment/scale --min=1 --max=3 --cpu-percent=80
horizontalpodautoscaler.autoscaling/scale autoscaled
```

8. To test your configuration, deploy the `stress` application to trigger the autoscaler.

`ApacheBench` (`ab`) is an Apache HTTP server benchmarking tool. The `stress` example application uses this tool to send many concurrent requests to your PHP application.

The `ab` command is called from the `DO101x-apps/stress/Dockerfile` file:

```
ab -dSrk -c 20 -n 50000000 http://${SCALE_SERVICE_HOST}:${SCALE_SERVICE_PORT}/
index.php
```

Notice the use of the `SCALE_SERVICE_HOST` and `SCALE_SERVICE_PORT` variables to refer to the `scale` application. RHOCP automatically sets those variables in all the pods in the project.

8.1. In the RHOCP web console, click the **Add** tab, and then click **Import from Git**.

8.2. Complete the form according to the following table. To access the Git parameters, click **Show Advanced Git Options**.

Stress Application Parameters

Parameter	Value
Git Repo URL	`https://github.com/yourgituser/DO101x-apps`
Git Reference	`scale`
Context Dir	`/stress`
Application	`Create application`
Application Name	`stress`
Name	`stress`

Make sure `Create a route to the application` is **not** checked.

To avoid unexpected errors, review the values that you entered in the form before proceeding. Click **Create** to start the build and deployment processes.

8.3. Wait for the `stress` application to deploy, and then consult the logs of the pod to confirm that the `ab` command is sending requests.

From the `Topology` page, click the **stress** icon, and then click the **Resources** tab. Click **View Logs** near the pod.

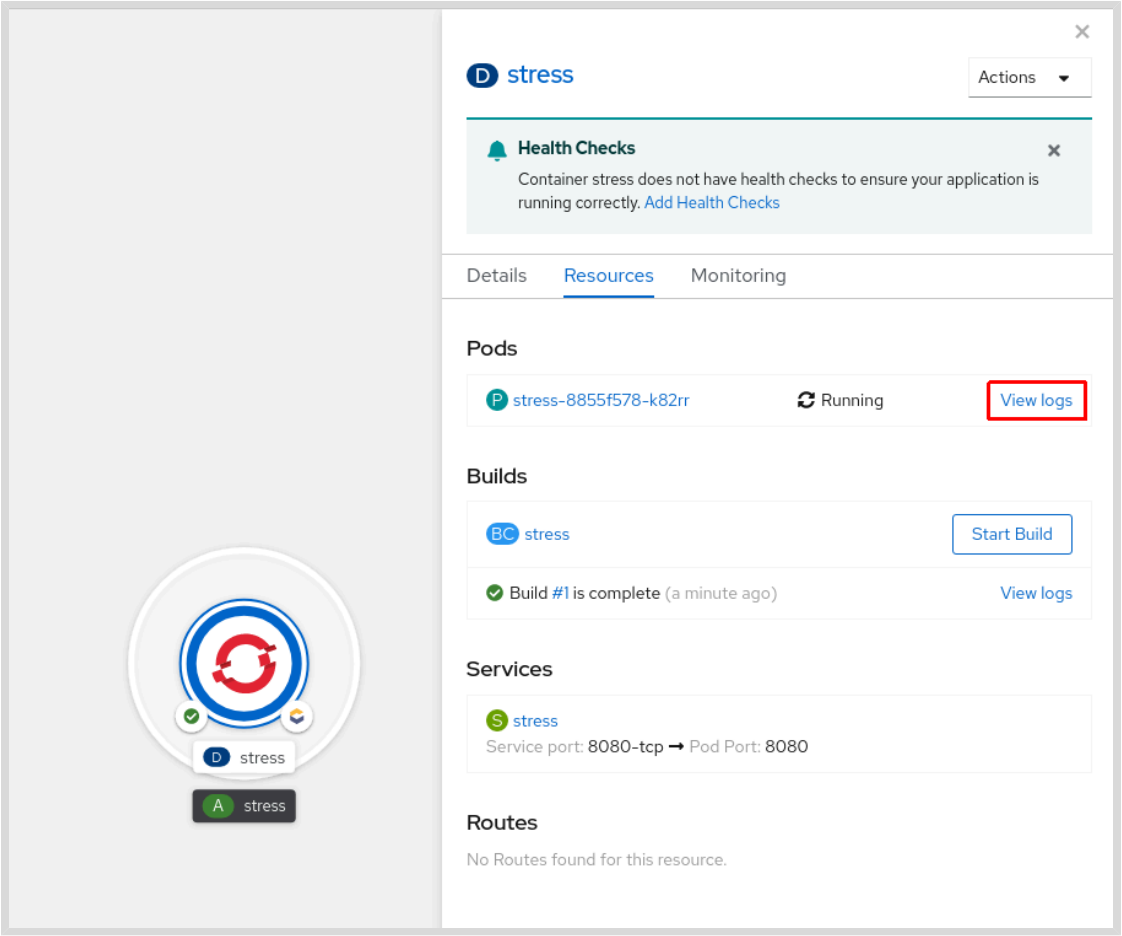


Figure 4.9: Accessing the logs of the stress pod

Notice that `ApacheBench` is running:

```
This is ApacheBench, Version 2.3 <$Revision: 1826891 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 172.30.199.15 (be patient)
```

The IP address displayed in the preceding output is probably different on your system.

9. Inspect the `scale` application and confirm that the number of pods automatically increases to three.

9.1. From the `Topology` page, click the PHP icon, and then click the **Details** tab.

9.2. Notice that the number of pods application increases to three.

Note
It might take a few minutes for the pods to increase.

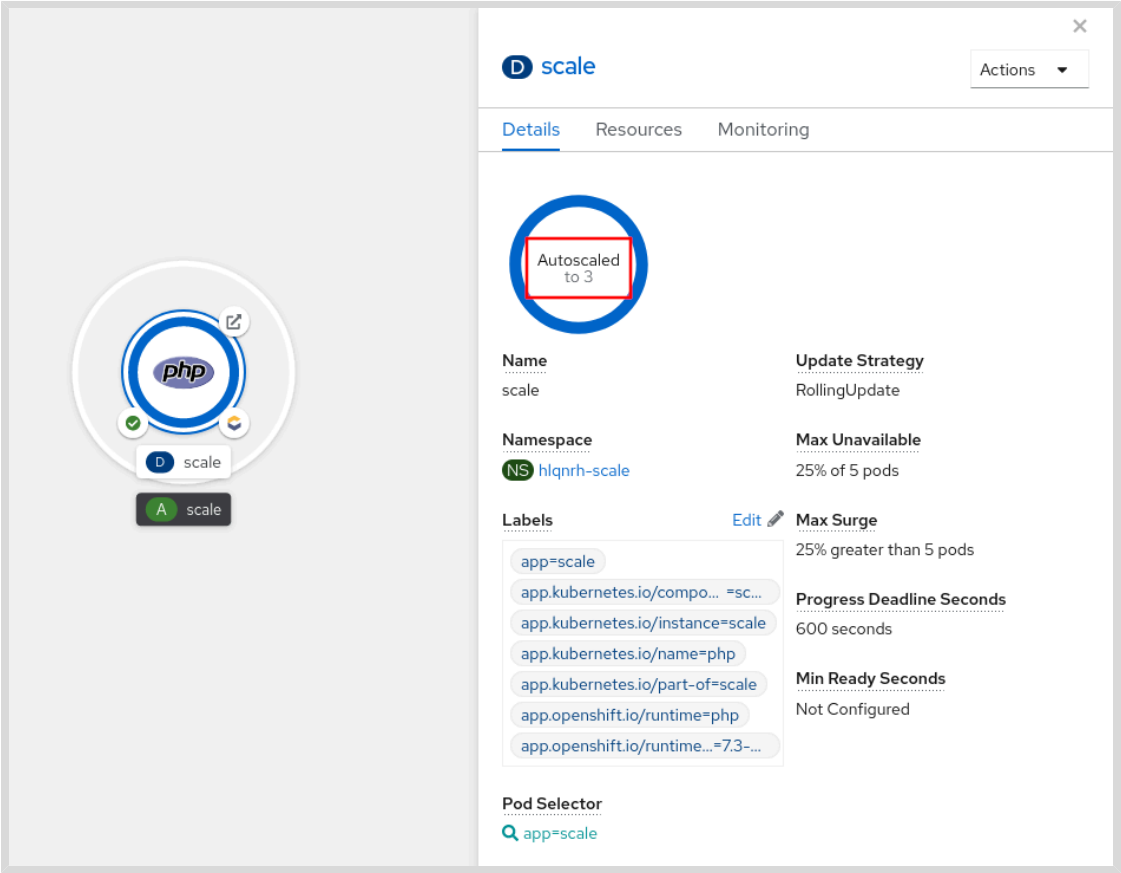


Figure 4.10: Autoscaling the application

10. Stop the `stress` application by manually scaling it down to zero pods. Notice that the `scale` application automatically scales down to one pod.

10.1. From the `Topology` page, click the **stress** icon, and then click the **Details** tab.

10.2. Click the down arrow to scale the application to zero pods.

10.3. Click the PHP icon, and then click the **Details** tab. It may take up to 10 minutes for the number of pods in the `scale` application to scale down to one.

11. Clean up.

11.1. RHOCP objects with the `app=stress` label:

```
$ oc delete all --selector app=stress
service "stress" deleted
deployment.apps "stress" deleted
buildconfig.build.openshift.io "stress" deleted
imagestream.image.openshift.io "stress" deleted
```

11.2. RHOCP objects with the `app=scale` label:

```
$ oc delete all --selector app=scale
pod "scale-9fd9d895c-h8wdm" deleted
service "scale" deleted
deployment.apps "scale" deleted
buildconfig.build.openshift.io "scale" deleted
imagestream.image.openshift.io "scale" deleted
route.route.openshift.io "scale" deleted
```

11.3. Delete the horizontal pod autoscaler object:

```
$ oc delete hpa scale
horizontalpodautoscaler.autoscaling "scale" deleted
```

11.4. Delete the `scale-generic-webhook-secret` and `scale-github-webhook-secret` secrets:

```
$ oc delete secret scale-generic-webhook-secret scale-github-webhook-secret
secret "scale-generic-webhook-secret" deleted
secret "scale-github-webhook-secret" deleted
```

11.5. Delete the `stress-generic-webhook-secret` and `stress-github-webhook-secret` secrets:

```
$ oc delete secret stress-generic-webhook-secret stress-github-webhook-secret
secret "stress-generic-webhook-secret" deleted
secret "stress-github-webhook-secret" deleted
```

Finish

This concludes the guided exercise.

Summary

In this chapter, you learned:

- Red Hat OpenShift Container Platform adapts your application to user demand by increasing or decreasing the number of pods.
- OpenShift automatically configures the route resource to load balance the user requests between pods.
- Use the OpenShift web console to scale applications manually.
- When configured, the Horizontal Pod Autoscaler (HPA) automatically scales applications based on CPU utilization.

Chapter 5

Troubleshooting Applications in OpenShift

Goal	Identify and resolve common problems in Red Hat OpenShift Container Platform.
Objectives	Troubleshoot an application by observing OpenShift logs and events.
Sections	Troubleshooting and Fixing an Application (and Guided Exercise)

Troubleshooting and Fixing an Application

Objectives

After completing this section, you should be able to troubleshoot an application by observing OpenShift logs and events.

Troubleshooting an Application with the OpenShift Web Console

The Source-to-Image (S2I) process is a simple way to automatically build and deploy an application from its source code.

This process is often a convenient way to deploy applications quickly. However, if either the build or the deployment operation fail, then you must troubleshoot and resolve any issues before the S2I process can successfully execute.

To identify and troubleshoot the error, it is helpful to understand that the S2I process is composed of two major steps:

- Build step – Compiles source code, downloads library dependencies, and packages the application as a container image. Red Hat OpenShift Container Platform uses the `BuildConfig` resource for the build step.

- Deployment step – Starts a pod and makes the application available. If the build step succeeds, then this step executes.

  OpenShift uses the `Deployment` or `DeploymentConfig` resources for the deployment step.

If you identify which step failed, then you will more easily identify why your application is not available as expected.

Inspecting Logs with the OpenShift Web Console

When an application fails, the web console helps developers identify the part of the deployment process in error. At each step, OpenShift maintains logs that developers can consult for troubleshooting.

You can consult the status of each application from the `Topology` page. The application icon provides a quick overview of its state.

A mark near the icon indicates the build status, as shown in the following screen capture. The icon on the left indicates a successful build. The icon on the right indicates a failed build.

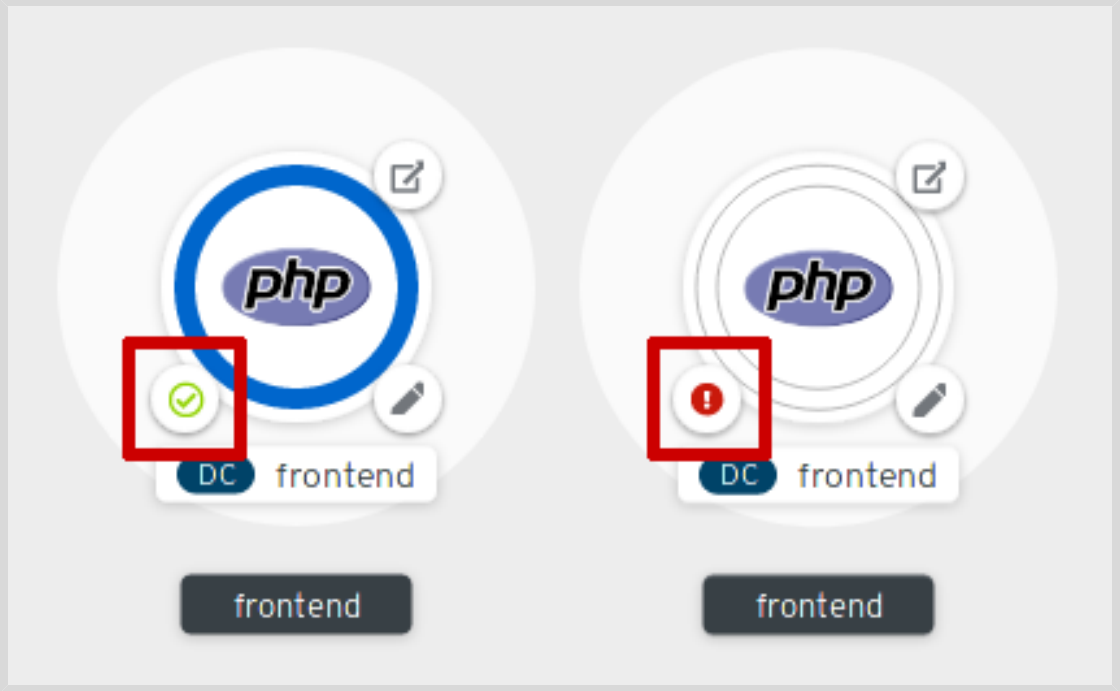


Figure 5.1: Successful and failed builds

If the build is successful, then OpenShift deploys the application. However, the deployment might also failed regardless of the build status. In the following screen capture, the icon on the left shows a successful build and a successful deployment. The icon on the right shows a successful build and a failed deployment.

Figure 5.2: Successful and failed deployments

To get more details and access the logs, click the application's icon and navigate to the `Resources` tab.

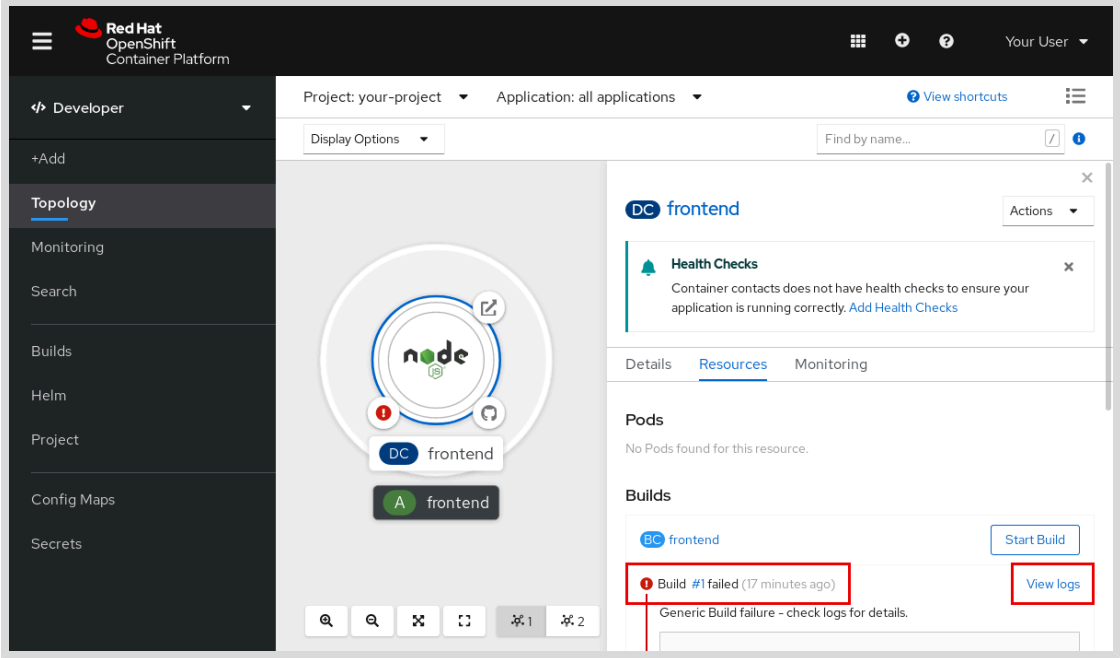


Figure 5.3: Details of a failed build

To identify any build issues, evaluate and analyze the logs for a build by clicking `View Logs`.

The detail page also provides access to the deployment logs.

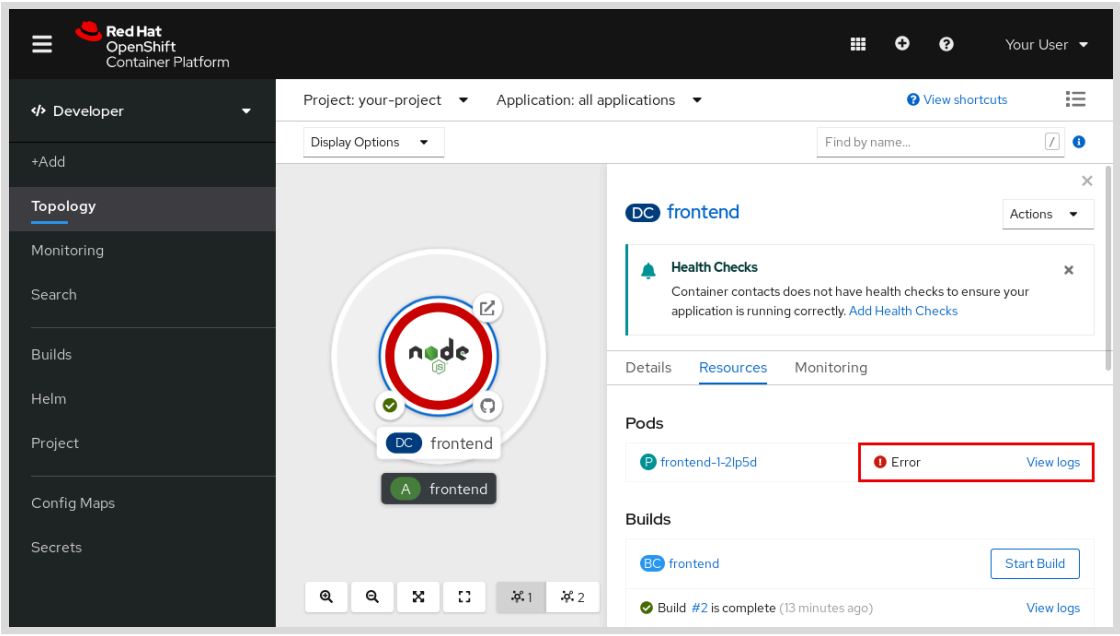


Figure 5.4: Details of a failed deployment

Accessing a Running Pod

Sometimes, OpenShift successfully builds and deploys the application, but due to bugs in the application source code the application does not behave as expected. In this situation, developers can access the logs of the running pod. For advanced troubleshooting, a console in the pod is available to run commands, and access the runtime environment inside the pod.

From the `Topology` page, click the pod name to access all of its parameters.

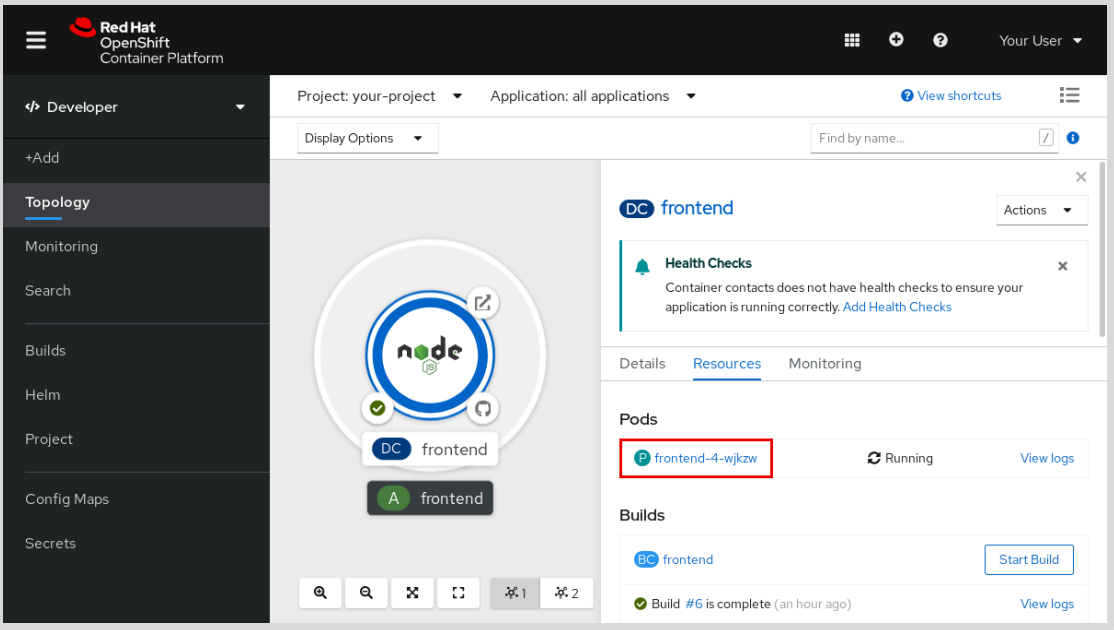


Figure 5.5: Accessing the pod details

The page that displays provides a tab to access the logs of the running pod. These logs show the output of the application that is running inside the pod.

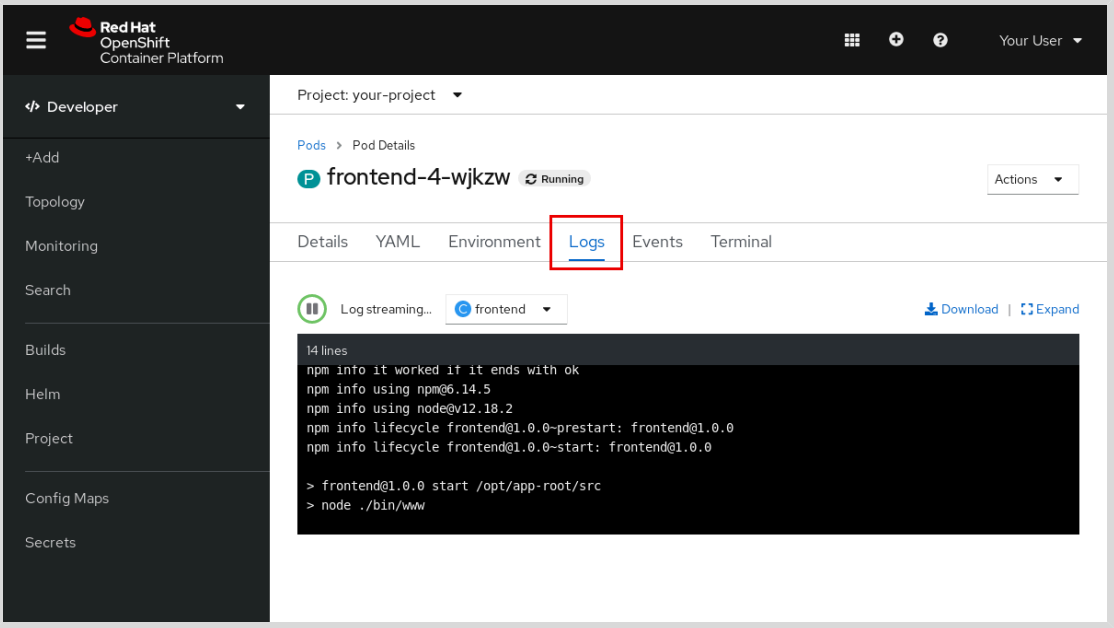


Figure 5.6: Consulting the pod logs

A console inside the container that is running the application is accessible on the `Terminal` tab. Expert developers can use this console for advanced troubleshooting.

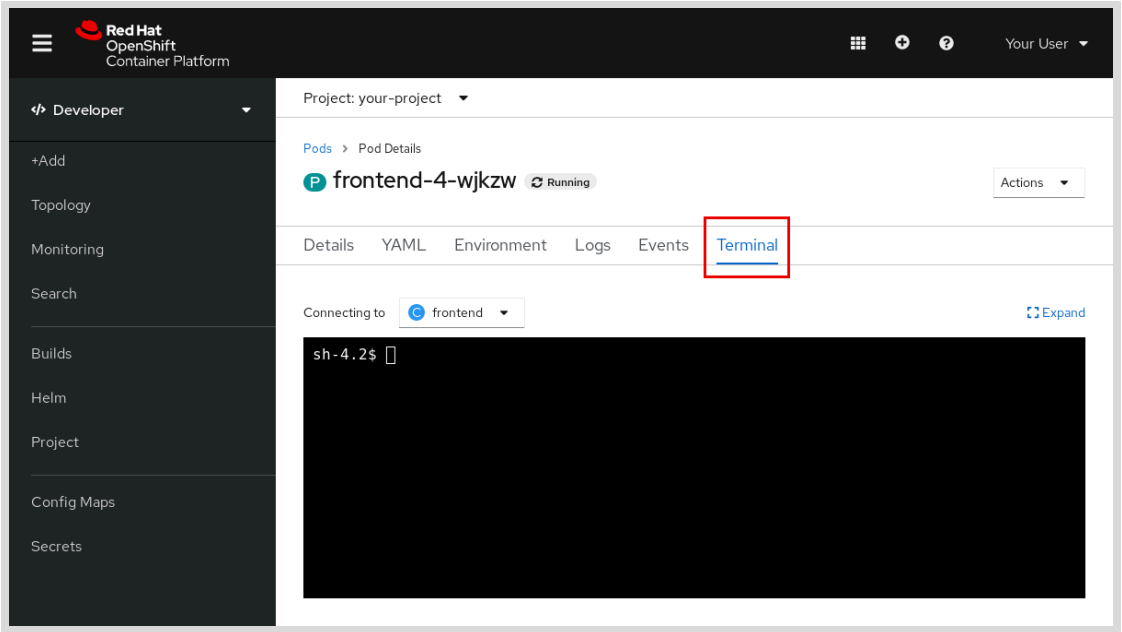


Figure 5.7: Accessing the pod terminal

Debugging an Application Running on OpenShift

For some languages, such as Node.js and Java, OpenShift provides features to attach a debugger to a running pod.

The debugger usually runs on a local workstation and connects to the application running on OpenShift through a remote debug port.

Remote debugging of an application is beyond the scope of this course. For more information, consult the following documentation:

- How to Debug Your Node.js Application on OpenShift with Chrome DevTools at https://developers.redhat.com/blog/2018/05/15/debug-your-node-js-application-on-openshift-with-chrome-devtools/
- Remote Debugging of Java Applications on OpenShift at https://servicesblog.redhat.com/2019/03/06/remote-debugging-of-java-applications-on-openshift/

Redeploying an Application after Fixing an Issue

After the source code has been fixed, committed, and pushed to the Git repository, developers can use the OpenShift web console to start a new build. When the build is complete, OpenShift automatically redeploys the application.

Remember, you can also configure webhooks for OpenShift to automatically start a new build with each commit. This way, as soon as the developer commits a fix, OpenShift redeploys the application without delay or manual intervention.

Accessing OpenShift Events

To simplify troubleshooting, OpenShift provides a high-level logging and auditing facility called `events`. OpenShift events signal significant actions, such as starting or destroying a pod.

To read the OpenShift events in the web console, select the **Home** > **Events** menu in the `Administrator` perspective.

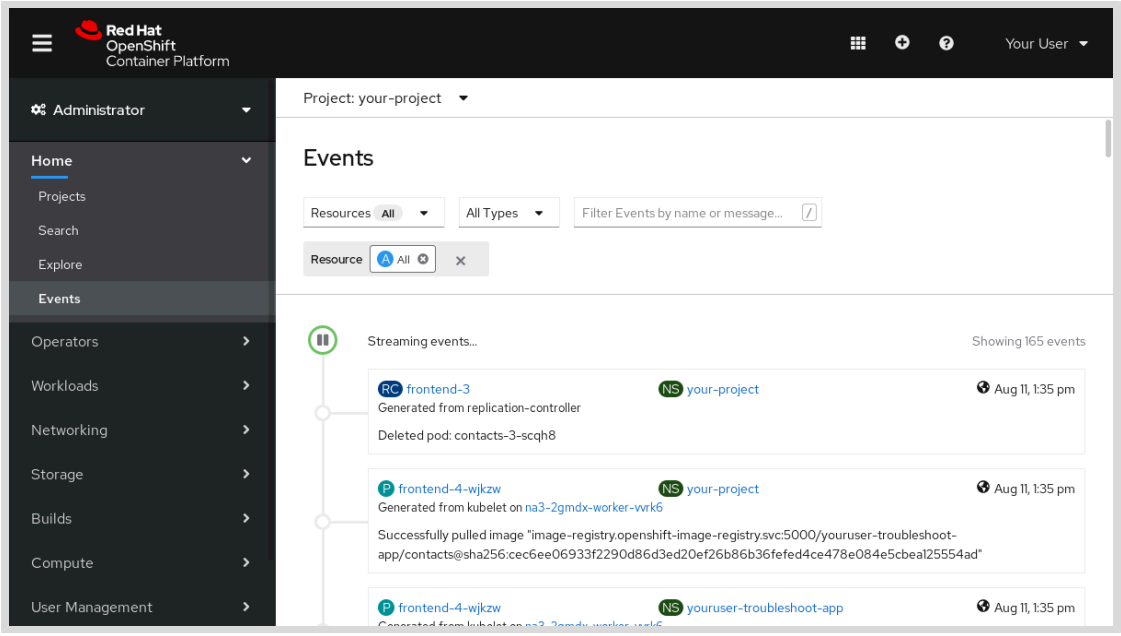


Figure 5.8: Listing the project events

From this page, developers have an overall view of the project events. Viewing the OpenShift events associated with a project is an important step to understanding what an application is doing.

Because OpenShift automatically refreshes the page with new events, developers can follow the progress of deployment in real time. In particular, when a build or an application deployment fails, there often are critical hints to the root cause of the problem provided in the project events. Sometimes, the provided information is enough for the developer to fix the application without having to inspect the more detailed OpenShift logs.

OpenShift uses events to report errors, such as when a build or a pod creation fails. It also notifies normal conditions through events, such as when OpenShift automatically scales an application by adding a new pod.

Troubleshooting Missing Environment Variables

Sometimes, the source code requires customization that is unavailable in containerized environments, such as database credentials, file system access, or message queue information. Those values are usually provided by leveraging environment variables within the pod.

For example, a MySQL instance will require certain environment variables to be configured in the environment prior to starting. If these variables are missing, the service will fail to start. Developers using the S2I process might need to access or manage this information if a service or application behaves unexpectedly.

The OpenShift logs can indicate missing values or options that must be enabled, incorrect parameters or flags, or environment incompatibilities.

Troubleshooting Invalid Parameters

Multi-tiered applications typically require sharing certain parameters, such as login credentials for a back-end database. As a developer, it is important to ensure that the same values for parameters reach all pods in the application.

For example, if a Node.js application runs in one pod, connected with another pod running a database, then make sure that the two pods use the same user name and password for the database. Usually, logs from the application pod provide a clear indication of these problems and how to solve them.

A good practice to centralize shared parameters is to store them in Configuration Map or in Secret resources. Remember that those resources can be injected into pods as environment variables, through the Deployment Configuration. Injecting the same Configuration Map or Secret resource into different pods ensures that not only the same environment variables are available, but also the same values.

References

For more information on events, refer to the Viewing system event information in an OpenShift Container Platform cluster chapter in the Configuring and managing nodes in OpenShift Container Platform guide at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/nodes/index#nodes-containers-events

For more information, refer to the Performing and interacting with builds in OpenShift Container Platform at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/builds/index

Guided Exercise

Troubleshooting and Fixing an Application

In this exercise, you will use OpenShift logs to troubleshoot an application.

Outcomes

You should be able to use the OpenShift web console to:

- Identify build and deployment issues.
- Analyze application source code and OpenShift logs to diagnose problems.
- Fix issues and redeploy applications.

Before You Begin

To perform this exercise, ensure you have:

- Access to a running Red Hat OpenShift Container Platform (RHOCP) cluster. See *Guided Exercise: Deploying an Application to Red Hat OpenShift Container Platform* for more information about using the developer sandbox RHOCP cluster.
- Visual Studio Code (VS Code) and Git are installed on your system
- Cloned your GitHub `DO101x-app` repository in VS Code.
- Installed the `oc` command.

Instructions

1. In VS Code, in the `DO101x-apps` repository, create a new branch named `troubleshoot` for recording your work in this exercise. Push that branch to your GitHub repository.

1.1. If you do not have VS Code open from a previous exercise, then open it.

1.2. In the source control view in VS Code (**View** > **SCM**), ensure that the `DO101x-apps` entry under `SOURCE CONTROL REPOSITORIES` shows the `main` branch.

Note

If the Source Control view does not display the `SOURCE CONTROL REPOSITORIES` heading, then right-click `SOURCE CONTROL` at the top of the Source Control view and select `Source Control Repositories`.

If necessary, click the current branch and select `main` in the `Select a ref to checkout` window to switch to the `main` branch.

> **Warning**
> Each exercise uses a unique branch. Always create a new branch using `main` as the base.

1.3. Click **main** in the `DO101x-apps` entry under `SOURCE CONTROL REPOSITORIES`, and then select `Create new branch`. Type `troubleshoot` to name the branch.

1.4. To push the new branch to GitHub, click the `Publish Changes` icon for the `DO101x-apps` entry. If prompted to authorize VS Code to push commits, then click **Allow** and continue on the GitHub website.

Alternatively, you can choose not to authorize GitHub. In such case, you will be prompted for username and password. Enter your GitHub developer token as your username, and leave the password field empty.

2. The `contacts-troubleshoot` application you deploy in this exercise requires a PostgreSQL database. Deploy that database from the catalog.

2.1. Log in to the RHOCP web console by using your developer account. Confirm the `Developer` perspective is active.

2.2. Click `Add`, and then click **Database**. Make sure that there are no filtering options selected.

2.3. Click **Postgres**, and then click `PostgreSQL (Ephemeral)`.

> **Warning**
> If you do not see the `PostgreSQL (Ephemeral)` template, make sure the `Template` filter is selected in the **Type** section.

Click **Instantiate Template**, and then complete the form according to the following table:

PostgreSQL Parameters

Parameter	Value
`Database Service Name`	`postgresql`
`PostgreSQL Database Name`	`contactsdb`

Leave the other fields with their default values and click **Create** to deploy the database. When deploying the database with these default parameters, OpenShift generates a random user name and password to access the database, and stores them in a secret resource.

2.4. To inspect the secret resource that OpenShift creates to store the database authentication parameters, click the `Secrets` menu.

2.5. From the resulting list of secrets, click the `postgresql` secret.

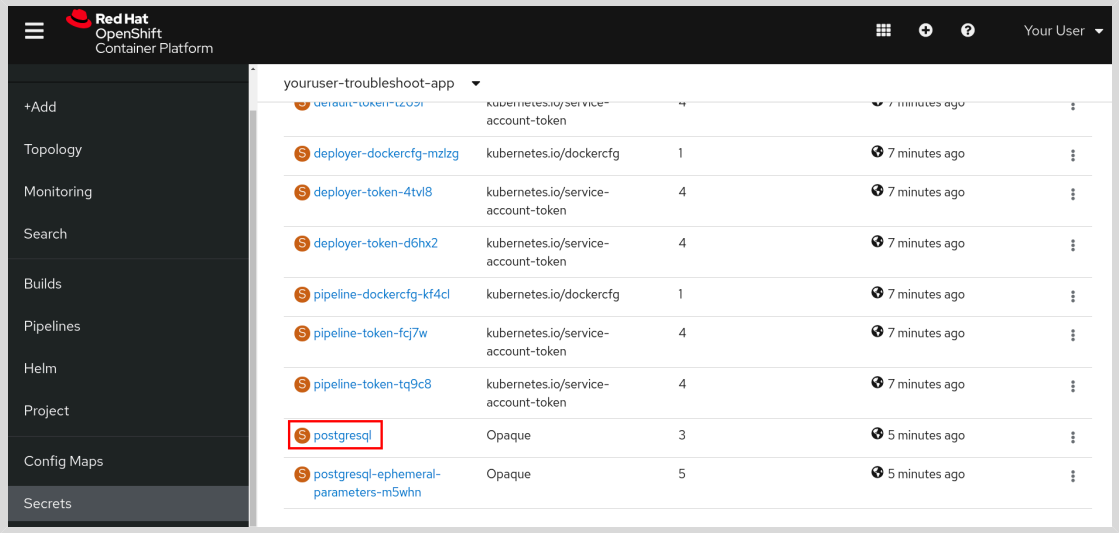


Figure 5.9: Accessing the secret resource

Under the `Data` section, notice that the secret stores the database name in the `database-name` entry, the password in the `database-password` entry, and the user name in the `database-user` entry.

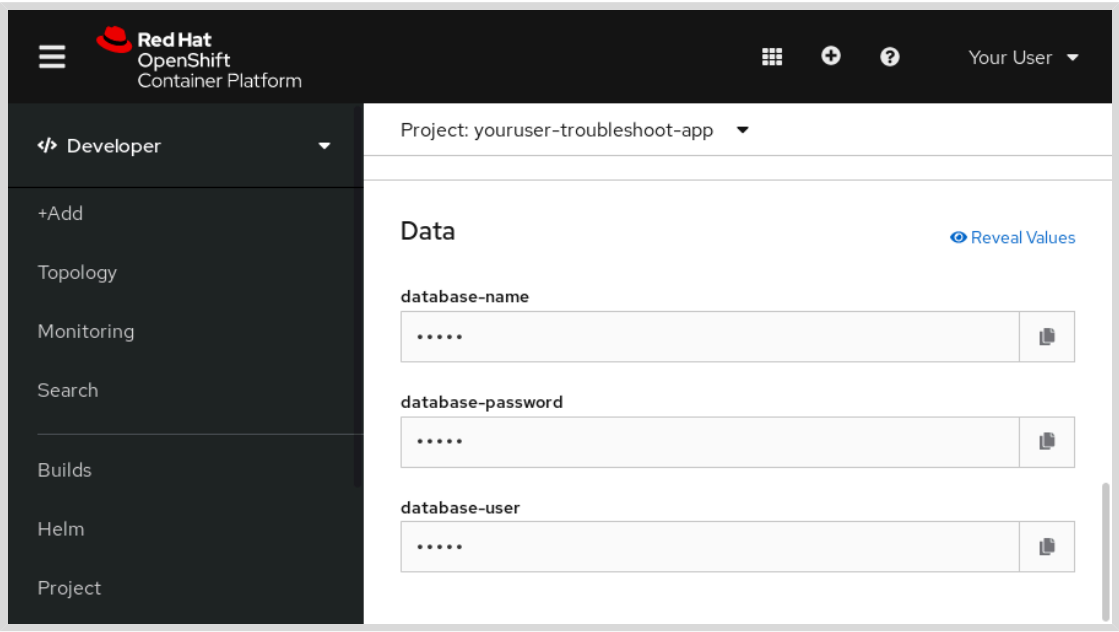


Figure 5.10: Data stored in the secret resource

3. Deploy the `contacts` JavaScript Node.js application. The code is in the `contacts-troubleshoot` subdirectory of your Git repository. Use the `troubleshoot` branch to deploy the application to OpenShift.

   3.1. Click **Add**, and then click **All services** in the **Developer Catalog** section.

   3.2. Click **Languages** > **JavaScript**, and then click the `Node.js` builder image. Click **Create Application** to enter the details of the application.

   3.3. Complete the form according to the following table. To access the Git parameters, click **Show Advanced Git Options**.

New Application Parameters

Parameter	Value
`Git Repo URL`	`https://github.com/yourgituser/DO101x-apps`
`Git Reference`	`troubleshoot`
`Context Dir`	`/contacts-troubleshoot`
`Application Name`	`contacts`
`Name`	`contacts`

To avoid unexpected errors when completing the following steps, review the values that you entered in the form before proceeding. Click **Create** to start the build and deployment processes.

4. OpenShift fails to build your application. Use the OpenShift web console to locate the error. Access the log to get more details.

   4.1. The web console should automatically show the `Topology` page. If this page is not displayed, then click the `Topology` tab.

   4.2. After a few minutes, the application displays an error state.

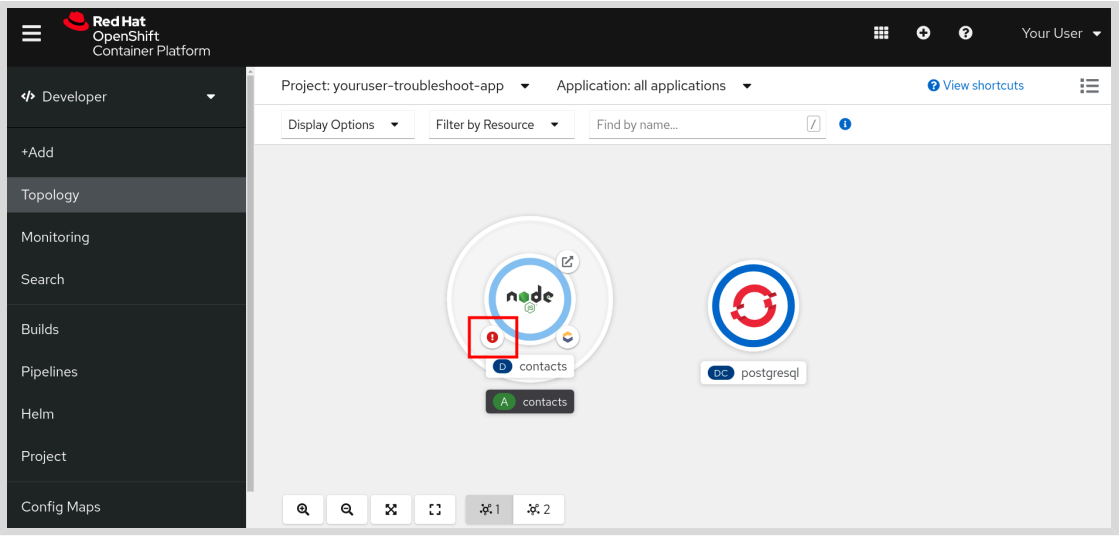


Figure 5.11: Error in the build step

Click the `Node.js` icon to access the application details (do not click the error check mark). Click the `Resources` tab to list the state of the resources.

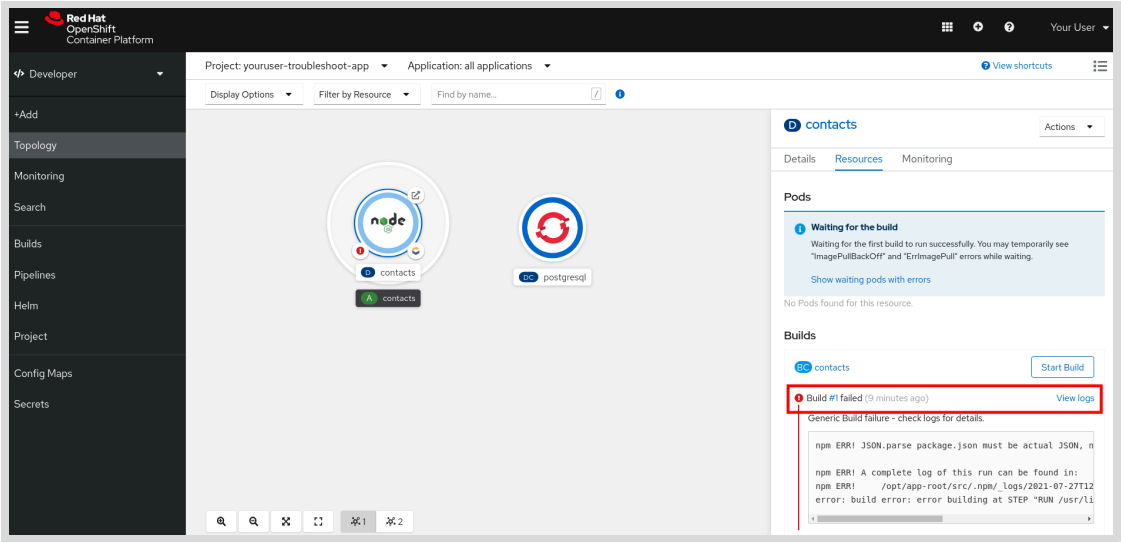


Figure 5.12: Accessing the error details

Notice that the build is in the error state. OpenShift is not able to deploy the application, and no pod is running.

4.3. Click `View Logs` to access the build log. Notice that the error occurs during the dependencies installation, and is caused by a syntax error in the `package.json` file.

```
...output omitted...
---> Installing dependencies
npm ERR! code EJSONPARSE
npm ERR! file /opt/app-root/src/package.json
npm ERR! JSON.parse Failed to parse json
npm ERR! JSON.parse Unexpected token d in JSON at position 118 while parsing near
 '... ./bin/www.js"
npm ERR! JSON.parse   },
npm ERR! JSON.parse dependencies": {
npm ERR! JSON.parse    ...'
npm ERR! JSON.parse Failed to parse package.json data.
npm ERR! JSON.parse package.json must be actual JSON, not just JavaScript.

npm ERR! A complete log of this run can be found in:
npm ERR!     /opt/app-root/src/.npm/_logs/2020-08-11T10_12_06_526Z-debug.log
subprocess exited with status 1
subprocess exited with status 1
error: build error: error building at STEP "RUN /usr/libexec/s2i/assemble": exit
 status 1
```

5. Open the `DO101x-apps/contacts-troubleshoot/package.json` file to identify and fix the issue. When finished, commit and push your change to GitHub.

5.1. In the VS Code Explorer view (**View** > **Explorer**), open the `DO101x-apps/contacts-troubleshoot/package.json` file.

5.2. The `dependencies` keyword must be enclosed in double quotes. Notice that the opening double quote is missing.

5.3. Add the missing double quote and then save the file.

```
"dependencies": {
   "cookie-parser": "1.4.5",
   "debug": "4.3.2",
   "dotenv": "10.0.0",
   "express": "4.17.1",
   "http-errors": "1.8.0",
   "morgan": "1.10.0",
   "pg": "8.6.0",
   "pug": "3.0.2"
},
```

5.4. Commit your change. From the Source Control view (**View** > **SCM**), click the + icon for the `package.json` entry to stage the file for the next commit. Click in the `Message (press Ctrl+Enter to commit)` field. Type `Fix syntax error` in the message box. Click the check mark icon to commit the change.

5.5. To push your change to the GitHub repository, click the `Synchronize Changes` icon for the `DO101x-apps` entry, under `SOURCE CONTROL REPOSITORIES`. If VS Code displays a message saying that this action will push and pull commits to and from `origin/troubleshoot`, then click **OK**.

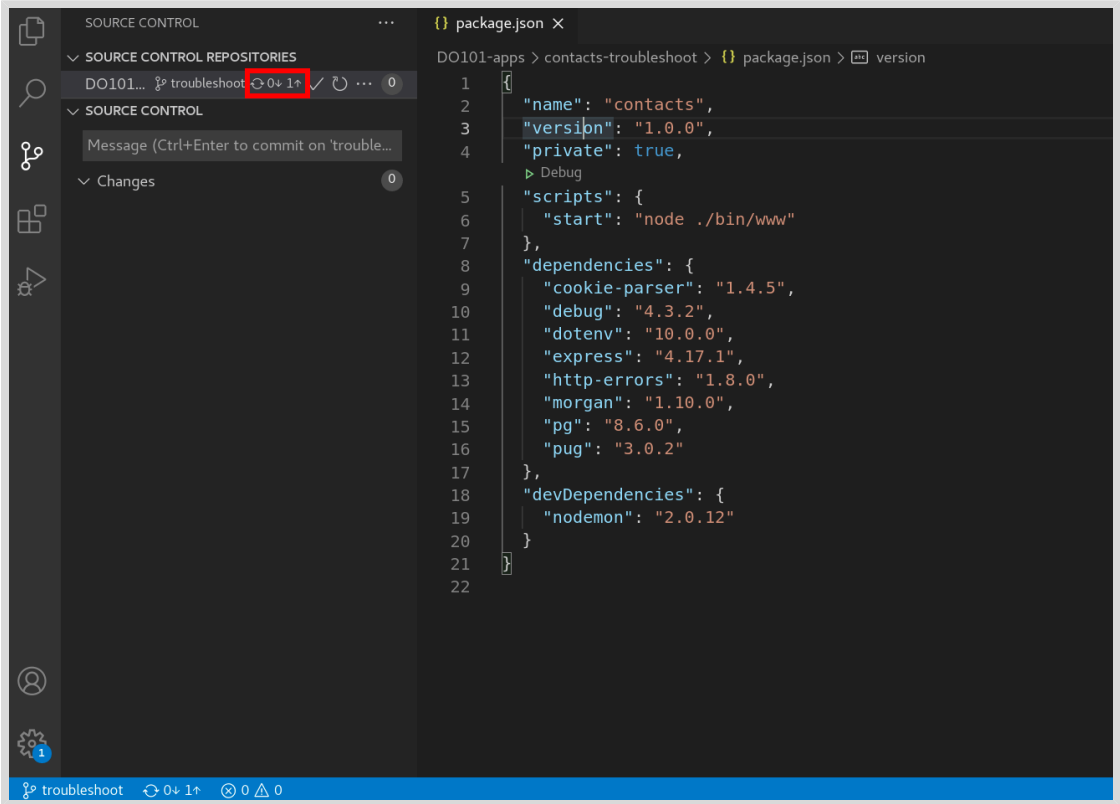


Figure 5.13: Pushing changes to GitHub

6. Use the OpenShift web console to initiate a new build and deployment. Click the `Topology` tab. Click the `Node.js` icon, and then click the `Resources` tab. Click **Start Build** to initiate a build. For this new build, OpenShift retrieves your fixed version of the application from GitHub.

7. OpenShift fails again to build or deploy your application. Use the OpenShift web console to locate the error. Access the log to get more details.

7.1. Wait for the build to finish and notice that the build is successful. The pod, however, is in the error state.

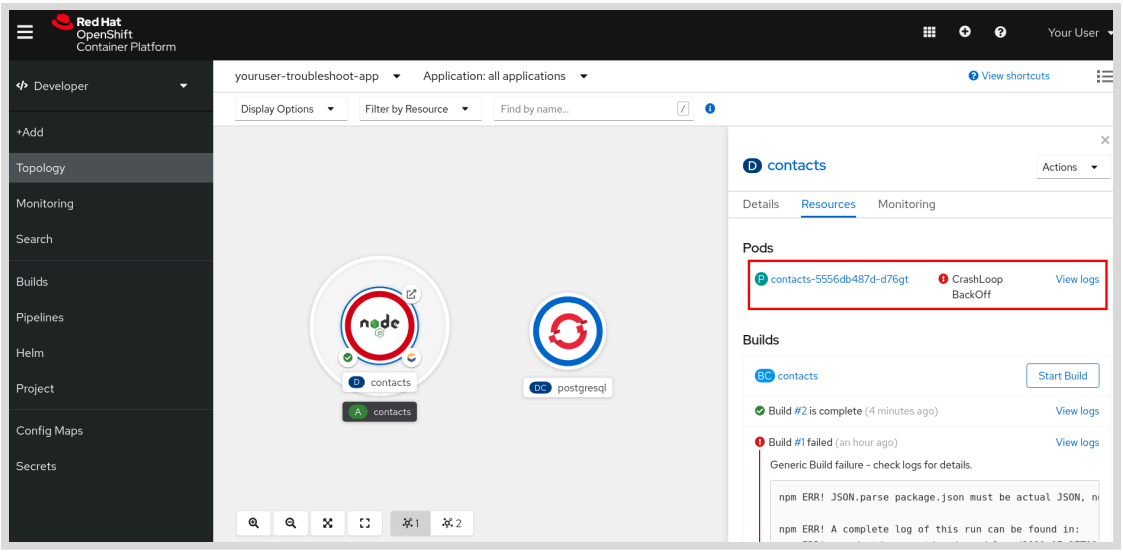


Figure 5.14: Deployment in error

The error status may alternate between `CrashLoopBackOff` and `Error`.

7.2. Click `View Logs` near the pod in error to access its log.

```
...output omitted...
> node ./bin/www.js

internal/modules/cjs/loader.js:638
    throw err;
    ^

Error: Cannot find module '/opt/app-root/src/bin/www.js'
...output omitted...
```

During startup, Node.js cannot find the program to run.

Note

If you have difficulties displaying the pod logs in the RHOCP web console, you can use the `oc` utility to get the logs.

Execute `oc get pods` to get the pod name. Execute `oc logs` *`POD_NAME`* to get the logs of a pod.

For example, `oc logs contacts-f45d565c9-8558b` prints the logs of the `contacts-f45d565c9-8558b` pod to the console.

8. The Node.js application declares the program to start in the `package.json` file. Identify and fix the issue in the `DO101x-apps/contacts-troubleshoot/package.json` file. When finished, commit and push your change to GitHub.

8.1. Open the `DO101x-apps/contacts-troubleshoot/package.json` file in VS Code.

8.2. The file defines the program to run in the `scripts` section. The program is `./bin/www.js`.

8.3. In VS Code, navigate to the `./bin/` directory (`DO101x-apps/contacts-troubleshoot/bin/`). In this directory, the program file is `www`, not `www.js`.

8.4. In `package.json`, replace `www.js` by `www`, and then save the file.

```
...output omitted...
"scripts": {
  "start": "node ./bin/www"
},
...output omitted...
```

8.5. Commit your change. From the Source Control view (**View** > **SCM**), click the + icon for the `package.json` entry to stage the file for the next commit. Click in the `Message (press Ctrl+Enter to commit)` field. Type `Fix wrong program name` in the message box. Click the check mark icon to commit the change.

8.6. To push your change to the GitHub repository, click the `Synchronize Changes` icon for the `DO101x-apps` entry, under `SOURCE CONTROL REPOSITORIES`.

9. Use the OpenShift web console to initiate a new build and deployment. Click the `Topology` tab. Click the `Node.js` icon and then the `Resources` tab. Click **Start Build** to start a build.

10. Wait a few minutes for the build and deployment to complete. This time, the build is successful and a pod is running. Access the application and notice the error message.

10.1. Click the **Open URL** button to access the application.

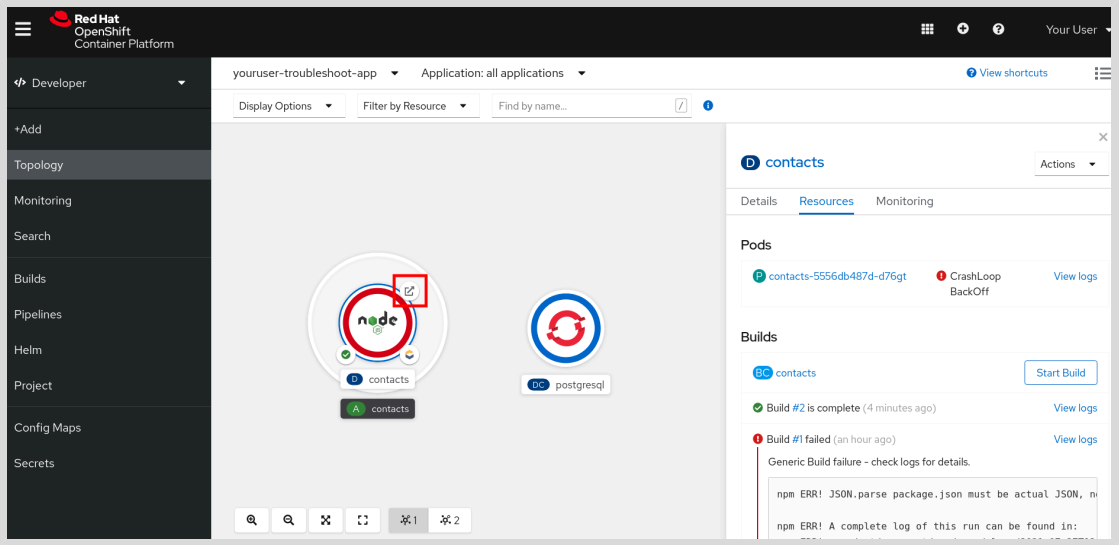


Figure 5.15: Accessing the contacts application

10.2. The application displays an error message regarding the database connection.

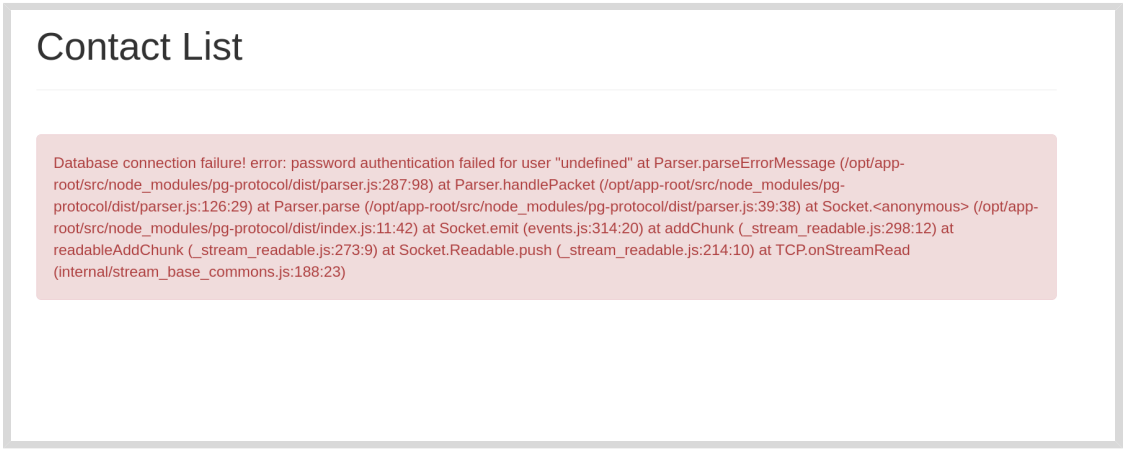


Figure 5.16: Database error

This message indicates that the provided user name for database authentication is not defined. When finished, close the browser tab.

11. The Node.js application defines the database parameters in the `DO101x-apps/contacts-troubleshoot/db/config.js` file. Open this file with VS Code.

```
...output omitted...
// The following variables should be defined in the
// secret resource associated with the database.
var db_user = process.env["database-user"];
var db_pass = process.env["database-password"];
var db_name = process.env["database-name"];
...output omitted...
```

The application retrieves the database connection parameters from the `database-user`, `database-password`, and `database-name` environment variables, but those variables are not available in the running pod.

12. Use the OpenShift web console to associate the `postgresql` secret to the `contacts` deployment configuration resource. With this association, OpenShift defines each entry in the secret as environment variables in the pod. When done, confirm that the application is finally working as expected.

12.1. Click the `contacts` deployment link.

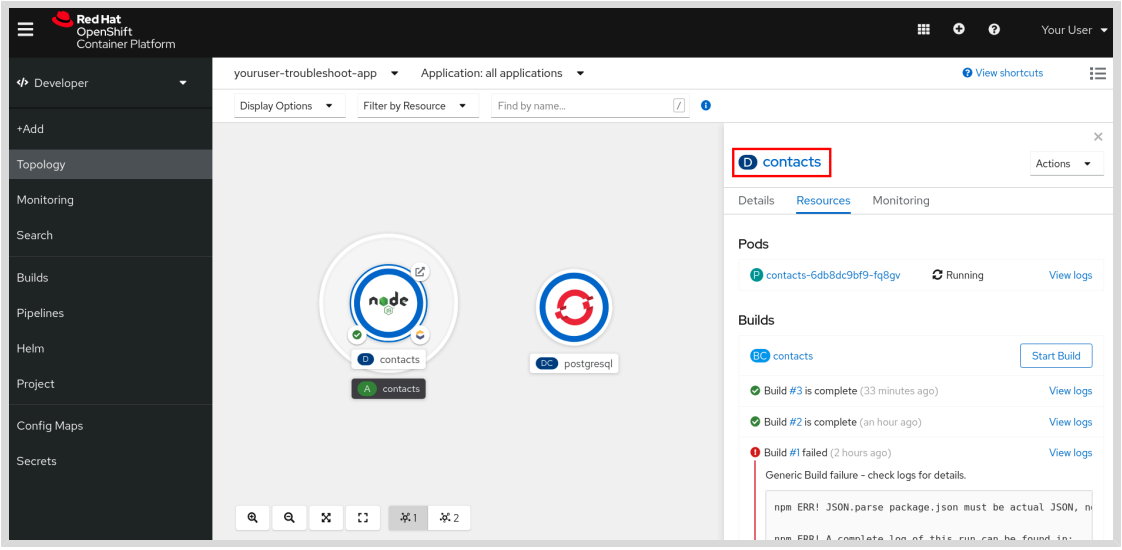


Figure 5.17: Accessing the deployment configuration details

12.2. Click the `Environment` tab. In the `All values from existing config maps or secrets (envFrom)` section, set the `Config Map/Secret` field to `postgresql`, and then click **Save**.

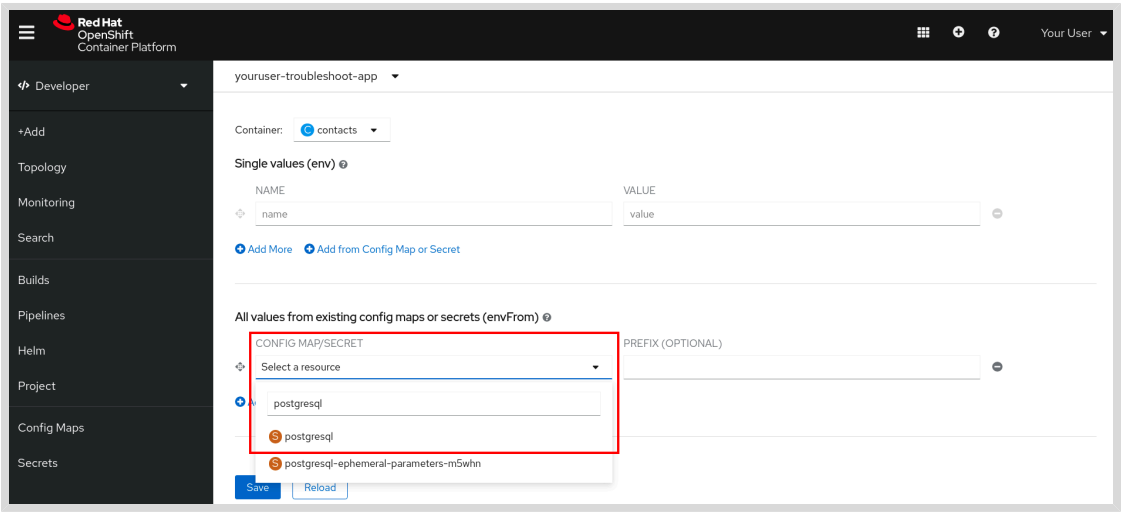


Figure 5.18: Associating the secret with the deployment configuration

When you update the deployment configuration resource, OpenShift automatically redeploys the application.

12.3. Wait a minute for the pod to redeploy and then test the application. To test the application, click `Topology`, and then click `Node.js` **Open URL**. The application displays the rows retrieved from the database.

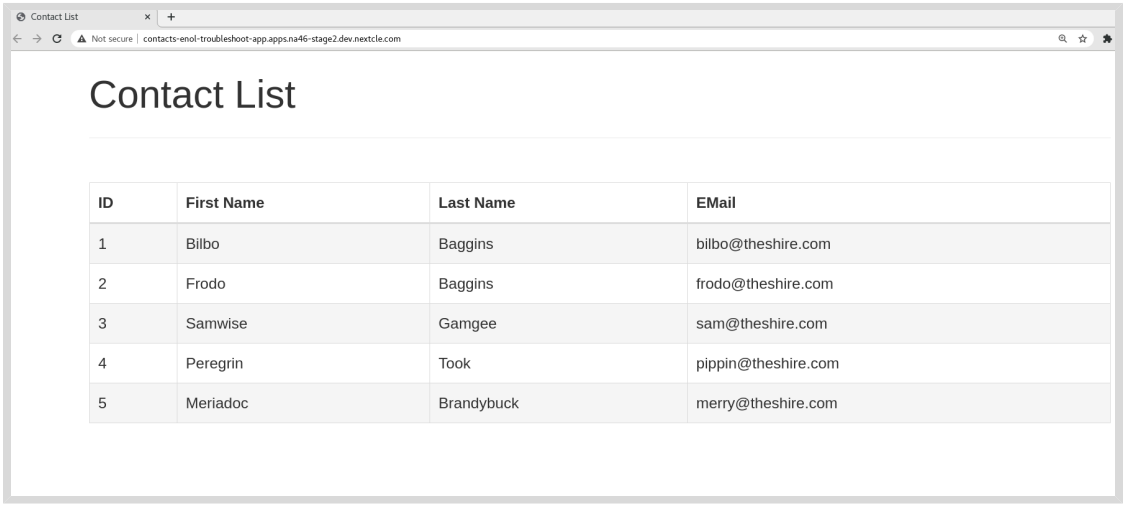


Figure 5.19: The contacts application

When done, close the browser tab.

13. Clean up.

13.1. Delete RHOCP objects with the `app=contacts` label:

```
$ oc delete all --selector app=contacts
pod "contacts-847b766bc8-f9bf8" deleted
service "contacts" deleted
deployment.apps "contacts" deleted
buildconfig.build.openshift.io "contacts" deleted
imagestream.image.openshift.io "contacts" deleted
route.route.openshift.io "contacts" deleted
```

13.2. Delete RHOCP objects with the `template=postgresql-ephemeral-template` label:

```
$ oc delete all,secret --selector template=postgresql-ephemeral-template
replicationcontroller "postgresql-1" deleted
service "postgresql" deleted
deploymentconfig.apps.openshift.io "postgresql" deleted
secret "postgresql" deleted
```

13.3. Delete the `contacts-github-webhook-secret`, and `contacts-generic-webhook-secret` secrets:

```
$ oc delete secret contacts-generic-webhook-secret contacts-github-webhook-secret
secret "contacts-generic-webhook-secret" deleted
secret "contacts-github-webhook-secret" deleted
```

Finish

This concludes the guided exercise.

Summary

In this chapter, you learned:

- The `Topology` page of the OpenShift web console is used to identify the deployment step in error.
- OpenShift maintains logs for the build, deployment, and running processes.
- For advanced troubleshooting, use the console to run commands inside a running pod.
- OpenShift events record significant actions, such as the start of a pod or the failure of a build process.

Chapter 6

Deploying Containerized Applications on OpenShift

Goal

Deploy single container applications on OpenShift Container Platform.

Objectives

- Describe the architecture of Kubernetes and Red Hat OpenShift Container Platform.
- Create standard Kubernetes resources.
- Expose services using OpenShift routes.
- Deploy an application using the Source-to-Image (S2I) facility of OpenShift Container Platform.

Sections

- Describing Kubernetes and OpenShift Architecture (and Quiz)
- Creating Kubernetes Resources (and Guided Exercise)
- Creating Routes (and Guided Exercise)
- Creating Applications with Source-to-Image (and Guided Exercise)

Describing Kubernetes and OpenShift Architecture

Objectives

After completing this section, students should be able to

- Describe the architecture of a Kubernetes cluster running on the Red Hat OpenShift Container Platform (RHOCP).
- List the main resource types provided by Kubernetes and RHOCP.
- Identify the network characteristics of containers, Kubernetes, and RHOCP.
- List mechanisms to make a pod externally available.

Kubernetes and OpenShift

In previous chapters we saw that Kubernetes is an orchestration service that simplifies the deployment, management, and scaling of containerized applications. One of the main advantages of using Kubernetes is that it uses several nodes to ensure the resiliency and scalability of its managed applications. Kubernetes forms a cluster of node servers that run containers and are centrally managed by a set of control plane servers. A server can act as both a control plane node and a compute node, but those roles are usually separated for increased stability.

Kubernetes Terminology

Term	Definition
Node	A server that hosts applications in a Kubernetes cluster.
Control Plane	Provides basic cluster services such as APIs or controllers.
Compute Node	This node executes workloads for the cluster. Application pods are scheduled onto compute nodes.
Resource	Resources are any kind of component definition managed by Kubernetes. Resources contain the configuration of the managed component (for example, the role assigned to a node), and the current state of the component (for example, if the node is available).
Controller	A controller is a Kubernetes process that watches resources and makes changes attempting to move the current state towards the desired state.
Label	A key-value pair that can be assigned to any Kubernetes resource. Selectors use labels to filter eligible resources for scheduling and other operations.
Namespace	A scope for Kubernetes resources and processes, so that resources with the same name can be used in different boundaries.

Note

The latest Kubernetes versions implement many controllers as Operators. Operators are Kubernetes plug-in components that can react to cluster events and control the state of resources. Operators and CoreOS Operator Framework are outside the scope of this document.

Red Hat OpenShift Container Platform is a set of modular components and services built on top of Red Hat CoreOS and Kubernetes. RHOCP adds PaaS capabilities such as remote management, increased security, monitoring and auditing, application lifecycle management, and self-service interfaces for developers.

An OpenShift cluster is a Kubernetes cluster that can be managed the same way, but using the management tools provided by OpenShift, such as the command-line interface or the web console. This allows for more productive workflows and makes common tasks much easier.

OpenShift Terminology

Term	Definition
Infra Node	A node server containing infrastructure services like monitoring, logging, or external routing.
Console	A web UI provided by the RHOCP cluster that allows developers and administrators to interact with cluster resources.
Project	OpenShift extension of Kubernetes' namespaces. Allows the definition of user access control (UAC) to resources.

The following schema illustrates the OpenShift Container Platform stack:

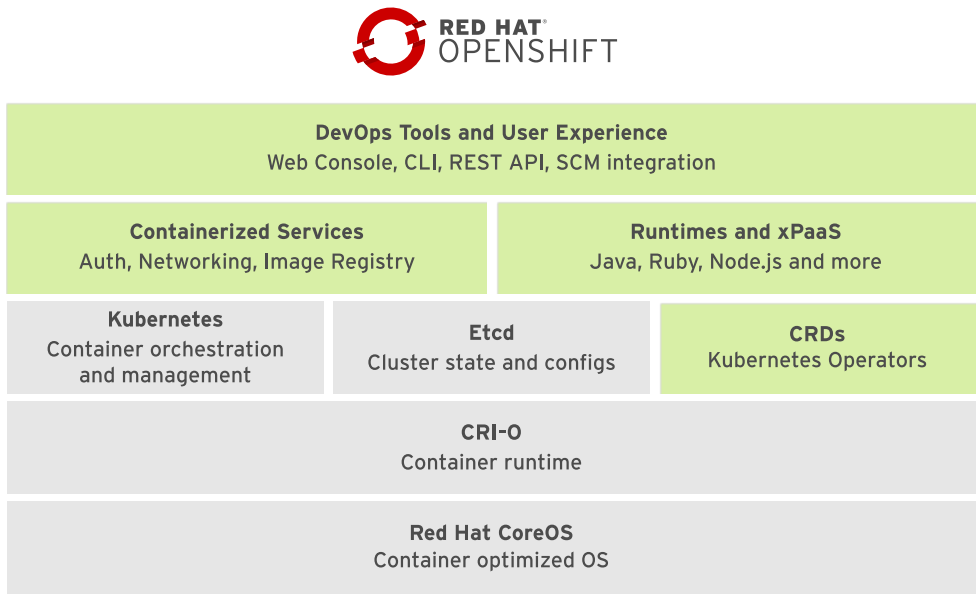


Figure 6.1: OpenShift component stack

From bottom to top, and from left to right, this shows the basic container infrastructure, integrated and enhanced by Red Hat:

- The base OS is Red Hat CoreOS. Red Hat CoreOS is a Linux distribution focused on providing an immutable operating system for container execution.

- CRI-O is an implementation of the Kubernetes Container Runtime Interface (CRI) to enable using Open Container Initiative (OCI) compatible runtimes. CRI-O can use any container runtime that satisfies CRI: `runc` (used by the Docker service), `libpod` (used by Podman) or `rkt` (from CoreOS).

- Kubernetes manages a cluster of hosts, physical or virtual, that run containers. It uses resources that describe multicontainer applications composed of multiple resources, and how they interconnect.

- Etcd is a distributed key-value store, used by Kubernetes to store configuration and state information about the containers and other resources inside the Kubernetes cluster.

- Custom Resource Definitions (CRDs) are resource types stored in Etcd and managed by Kubernetes. These resource types form the state and configuration of all resources managed by OpenShift.

- Containerized services fulfill many PaaS infrastructure functions, such as networking and authorization. RHOCP uses the basic container infrastructure from Kubernetes and the underlying container runtime for most internal functions. That is, most RHOCP internal services run as containers orchestrated by Kubernetes.

- Runtimes and xPaaS are base container images ready for use by developers, each preconfigured with a particular runtime language or database. The xPaaS offering is a set of base images for Red Hat middleware products such as JBoss EAP and ActiveMQ. Red Hat OpenShift Application Runtimes (RHOAR) are a set runtimes optimized for cloud native applications in OpenShift. The application runtimes available are Red Hat JBoss EAP, OpenJDK, Thorntail, Eclipse Vert.x, Spring Boot, and Node.js.

- RHOCP provides web UI and CLI management tools for managing user applications and RHOCP services. The OpenShift web UI and CLI tools are built from REST APIs which can be used by external tools such as IDEs and CI platforms.

This OpenShift and Kubernetes architecture illustration gives further insight into how the infrastructure components work together.

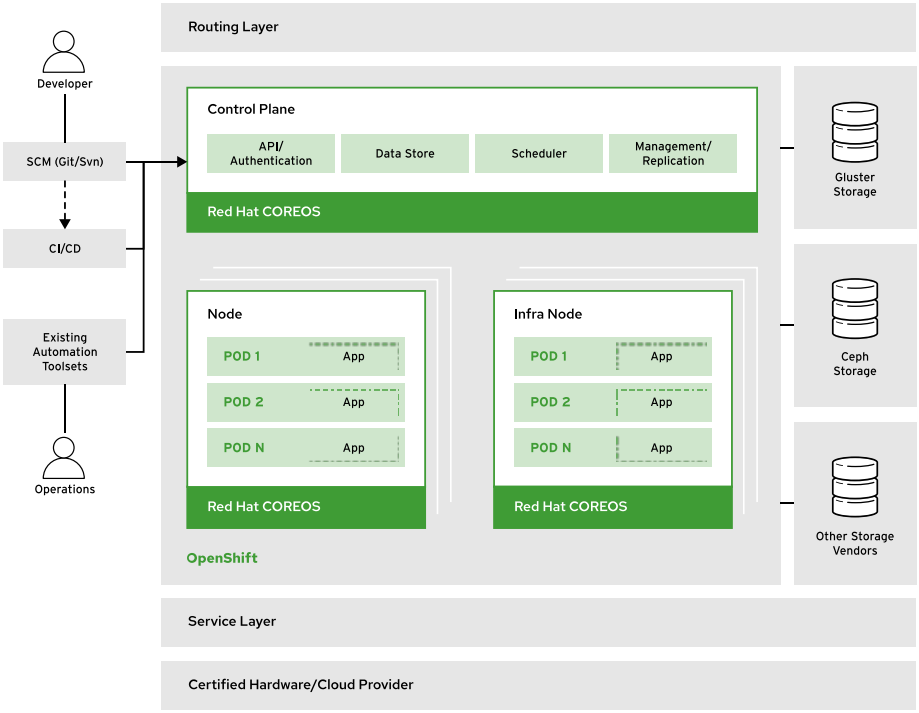


Figure 6.2: OpenShift and Kubernetes architecture

New Features in RHOCP 4

RHOCP 4 is a massive change from previous versions. As well as keeping backwards compatibility with previous releases, it includes new features, such as:

- CoreOS as the mandatory operating system for all nodes, offering an immutable infrastructure optimized for containers.
- A brand new cluster installer which guides the process of installation and update.
- A self-managing platform, able to automatically apply cluster updates and recoveries without disruption.
- A redesigned application lifecycle management.
- An Operator SDK to build, test, and package Operators.

Describing Kubernetes Resource Types

Kubernetes has six main resource types that can be created and configured using a YAML or a JSON file, or using OpenShift management tools:

Pods (po**)**
Represent a collection of containers that share resources, such as IP addresses and persistent storage volumes. It is the basic unit of work for Kubernetes.

Services (svc**)**
Define a single IP/port combination that provides access to a pool of pods. By default, services connect clients to pods in a round-robin fashion.

Replication Controllers (`rc`)
A Kubernetes resource that defines how pods are replicated (horizontally scaled) into different nodes. Replication controllers are a basic Kubernetes service to provide high availability for pods and containers.

Persistent Volumes (`pv`)
Define storage areas to be used by Kubernetes pods.

Persistent Volume Claims (`pvc`)
Represent a request for storage by a pod. PVCs links a PV to a pod so its containers can make use of it, usually by mounting the storage into the container's file system.

ConfigMaps (`cm`) and Secrets
Contains a set of keys and values that can be used by other resources. ConfigMaps and Secrets are usually used to centralize configuration values used by several resources. Secrets differ from ConfigMaps maps in that Secrets' values are always encoded (not encrypted) and their access is restricted to fewer authorized users.

Note
Although Kubernetes pods can be created standalone, they are usually created by high-level resources such as replication controllers.

OpenShift Resource Types

The main resource types added by OpenShift Container Platform to Kubernetes are as follows:

Deployment and Deployment config (`dc`)
OpenShift 4.5 introduced the Deployment resource concept to replace the DeploymentConfig as the default configuration for pods. Both are the representation of a set of containers included in a pod, and the deployment strategies to be used. It contains the configuration to be applied to all containers of each pod replica, such as the base image, tags, storage definitions and the commands to be executed when the containers start.

The `Deployment` object serves as the improved version of the `DepoymentConfig` object. Some replacements of functionalities between both objects are the following:

- Automatic rollback is no longer supported by deployment objects.
- Every change in the pod template used by deployment objects triggers a new rollout automatically.
- Lifecycle hooks are no longer supported by deployment objects.
- The deployment process of a deployment object can be paused at any time without affecting the deployer process.
- A deployment object can have as many active replica sets as the user wants, scaling down old replicas after a while. In contrast, the deploymentconfig object can only have two replication sets active at the same time.

Even if `Deployment` objects are meant to act as the default replacement of `DeploymentConfig` objects, in OpenShift 4.6 users can still make use of them if they need a specific feature provided by these objects. In this case, it is necessary to specify the type of object when creating a new application by specifying the `--as-deployment-config` flag.

Build config (`bc`)
Defines a process to be executed in the OpenShift project. Used by the OpenShift Source-to-Image (S2I) feature to build a container image from application source code stored in a Git repository. A `bc` works together with a `dc` to provide a basic but extensible continuous integration and continuous delivery workflows.

Routes
Represent a DNS host name recognized by the OpenShift router as an ingress point for applications and microservices.

Note
To obtain a list of all the resources available in a RHOCP cluster and their abbreviations, use the `oc api-resources` or `kubectl api-resources` commands.

Although Kubernetes replication controllers can be created standalone in OpenShift, they are usually created by higher-level resources such as deployment controllers.

Describing Networking in Red Hat OpenShift

Each container deployed in a Kubernetes cluster has an IP address assigned from an internal network that is accessible only from the node running the container. Because of the container's ephemeral nature, IP addresses are constantly assigned and released.

Kubernetes provides a software-defined network (SDN) that spawns the internal container networks from multiple nodes and allows containers from any pod, inside any host, to access pods from other hosts. Access to the SDN only works from inside the same Kubernetes cluster.

Containers inside Kubernetes pods should not connect to each other's dynamic IP address directly. Services resolves this problem by linking more stable IP addresses from the SDN to the pods. If pods are restarted, replicated, or rescheduled to different nodes, services are updated, providing scalability and fault tolerance.

External access to containers is more complicated. Kubernetes services can be set as a `NodePort` service type. With this type of service, Kubernetes allocates a network port from a predefined range in each of the cluster nodes and proxies it into your service. Unfortunately, this approach does not scale well.

OpenShift makes external access to containers both scalable and simpler by defining route resources. A route defines external-facing DNS names and ports for a service. A router (ingress controller) forwards HTTP and TLS requests to the service addresses inside the Kubernetes SDN. The only requirement is that the desired DNS names are mapped to the IP addresses of the RHOCP router nodes.

References

Kubernetes documentation website
https://kubernetes.io/docs/

OpenShift documentation website
https://docs.openshift.com/

Understanding Operators
https://docs.openshift.com/container-platform/4.6/operators/olm-what-operators-are.html

Creating Kubernetes Resources

Objectives

After completing this section, students should be able to create standard Kubernetes resources.

The Red Hat OpenShift Container Platform (RHOCP) Command-line Tool

The main method of interacting with an RHOCP cluster is using the `oc` command. The basic usage of the command is through its subcommands in the following syntax:

```
$ oc <command>
```

Before interacting with a cluster, most operations require the user to log in. The syntax to log in is shown below:

```
$ oc login <clusterUrl>
```

Describing Pod Resource Definition Syntax

RHOCP runs containers inside Kubernetes pods, and to create a pod from a container image, OpenShift needs a *pod resource definition*. This can be provided either as a JSON or YAML text file, or can be generated from defaults by the `oc new-app` command or the OpenShift web console.

A pod is a collection of containers and other resources. An example of a WildFly application server pod definition in YAML format is shown below:

```
apiVersion: v1
kind: Pod(1)
metadata:
  name: wildfly(2)
  labels:
    name: wildfly(3)
spec:
  containers:
    - resources:
        limits:
          cpu: 0.5
      image: do276/todojee(4)
      name: wildfly
      ports:
        - containerPort: 8080(5)
          name: wildfly
      env:(6)
        - name: MYSQL_ENV_MYSQL_DATABASE
          value: items
```

```
        - name: MYSQL_ENV_MYSQL_USER
          value: user1
        - name: MYSQL_ENV_MYSQL_PASSWORD
          value: mypa55
```

1. Declares a Kubernetes pod resource type.
2. A unique name for a pod in Kubernetes that allows administrators to run commands on it.
3. Creates a label with a key named `name` that other resources in Kubernetes, usually as service, can use to find it.
4. Defines the container image name.
5. A container-dependent attribute identifying which port on the container is exposed.
6. Defines a collection of environment variables.

Some pods may require environment variables that can be read by a container. Kubernetes transforms all the `name` and `value` pairs to environment variables. For instance, the `MYSQL_ENV_MYSQL_USER` variable is declared internally by the Kubernetes runtime with a value of `user1`, and is forwarded to the container image definition. Because the container uses the same variable name to get the user's login, the value is used by the WildFly container instance to set the username that accesses a MySQL database instance.

Describing Service Resource Definition Syntax

Kubernetes provides a virtual network that allows pods from different compute nodes to connect. But, Kubernetes provides no easy way for a pod to discover the IP addresses of other pods:

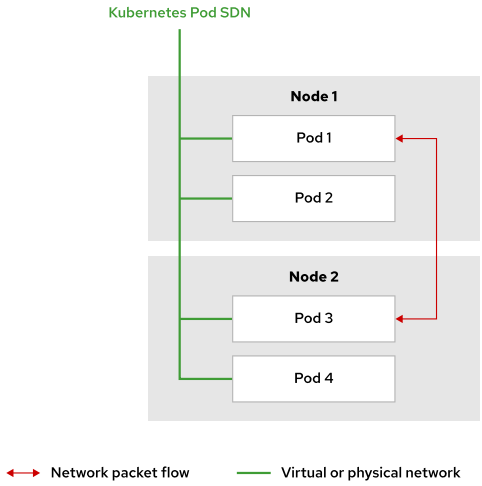


Figure 6.3: Basic Kubernetes networking

If Pod 3 fails and is restarted, it could return with a different IP address. This would cause Pod 1 to fail when attempting to communicate with Pod 3. A service layer provides the abstraction required to solve this problem.

Services are essential resources to any OpenShift application. They allow containers in one pod to open network connections to containers in another pod. A pod can be restarted for many reasons, and it gets a different internal IP address each time. Instead of a pod having to discover the IP

address of another pod after each restart, a service provides a stable IP address for other pods to use, no matter what compute node runs the pod after each restart:

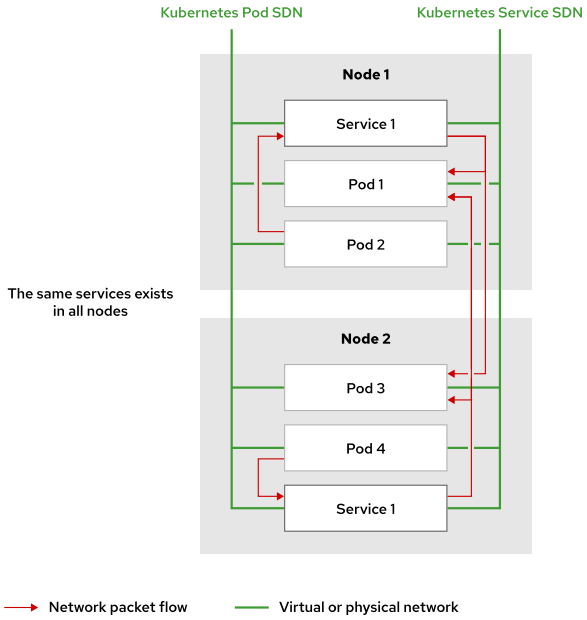


Figure 6.4: Kubernetes services networking

Most real-world applications do not run as a single pod. They need to scale horizontally, so many pods run the same containers from the same pod resource definition to meet growing user demand. A service is tied to a set of pods, providing a single IP address for the whole set, and a load-balancing client request among member pods.

The set of pods running behind a service is managed by a Deployment resource. A Deployment resource embeds a ReplicationController that manages how many pod copies (replicas) have to be created, and creates new ones if any of them fail. Deployment and ReplicationController resources are explained later in this chapter.

The following example shows a minimal service definition in JSON syntax:

```
{
    "kind": "Service", (1)
    "apiVersion": "v1",
    "metadata": {
        "name": "quotedb" (2)
    },
    "spec": {
        "ports": [ (3)
            {
                "port": 3306,
                "targetPort": 3306
            }
        ],
        "selector": {
            "name": "mysqldb" (4)
```

```
        }
    }
}
```

1. The kind of Kubernetes resource. In this case, a Service.
2. A unique name for the service.
3. `ports` is an array of objects that describes network ports exposed by the service. The `targetPort` attribute has to match a `containerPort` from a pod container definition, and the `port` attribute is the port that is exposed by the service. Clients connect to the service port and the service forwards packets to the pod `targetPort`.
4. `selector` is how the service finds pods to forward packets to. The target pods need to have matching labels in their metadata attributes. If the service finds multiple pods with matching labels, it load balances network connections between them.

Each service is assigned a unique IP address for clients to connect to. This IP address comes from another internal OpenShift SDN, distinct from the pods' internal network, but visible only to pods. Each pod matching the `selector` is added to the service resource as an endpoint.

Discovering Services

An application typically finds a service IP address and port by using environment variables. For each service inside an OpenShift project, the following environment variables are automatically defined and injected into containers for all pods inside the same project:

- *SVC_NAME*`_SERVICE_HOST` is the service IP address.
- *SVC_NAME*`_SERVICE_PORT` is the service TCP port.

Note

The *SVC_NAME* part of the variable is changed to comply with DNS naming restrictions: letters are capitalized and underscores (_) are replaced by dashes (-).

Another way to discover a service from a pod is by using the OpenShift internal DNS server, which is visible only to pods. Each service is dynamically assigned an SRV record with an FQDN of the form:

```
SVC_NAME.PROJECT_NAME.svc.cluster.local
```

When discovering services using environment variables, a pod has to be created and started only after the service is created. If the application was written to discover services using DNS queries, however, it can find services created after the pod was started.

There are two ways for an application to access the service from outside an OpenShift cluster:

1. `NodePort` type: This is an older Kubernetes-based approach, where the service is exposed to external clients by binding to available ports on the compute node host, which then proxies connections to the service IP address. Use the `oc edit svc` command to edit service attributes and specify `NodePort` as the value for `type`, and provide a port value for the `nodePort` attribute. OpenShift then proxies connections to the service via the public IP address of the compute node host and the port value set in `NodePort`.

2. OpenShift Routes: This is the preferred approach in OpenShift to expose services using a unique URL. Use the `oc expose` command to expose a service for external access or expose a service from the OpenShift web console.

Figure 6.5 illustrates how `NodePort` services allow external access to Kubernetes services. OpenShift routes are covered in more detail later in this course.

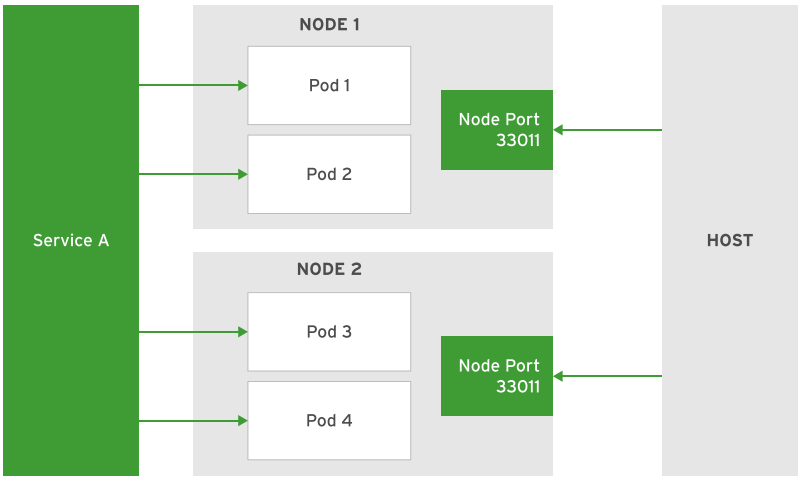


Figure 6.5: Alternative method for external access to a Kubernetes service

OpenShift provides the `oc port-forward` command for forwarding a local port to a pod port. This is different from having access to a pod through a service resource:

- The port-forwarding mapping exists only on the workstation where the `oc` client runs, while a service maps a port for all network users.
- A service load balances connections to potentially multiple pods, whereas a port-forwarding mapping forwards connections to a single pod.

Note

Red Hat discourages the use of the `NodePort` approach to avoid exposing the service to direct connections. Mapping via port-forwarding in OpenShift is considered a more secure alternative.

The following example demonstrates the use of the `oc port-forward` command:

```
[user@host ~]$ oc port-forward mysql-openshift-1-glqrp 3306:3306
```

The command forwards port 3306 from the developer machine to port 3306 on the `db` pod, where a MySQL server (inside a container) accepts network connections.

Note

When running this command, make sure you leave the terminal window running. Closing the window or canceling the process stops the port mapping.

Creating Applications

Simple applications, complex multitier applications, and microservice applications can be described by a single resource definition file. This single file would contain many pod definitions, service definitions to connect the pods, replication controllers or Deployment to horizontally scale the application pods, PersistentVolumeClaims to persist application data, and anything else needed that can be managed by OpenShift.

The `oc new-app` command can be used with the `-o json` or `-o yaml` option to create a skeleton resource definition file in JSON or YAML format, respectively. This file can be customized and used to create an application using the `oc create -f <filename>` command, or merged with other resource definition files to create a composite application.

The `oc new-app` command can create application pods to run on OpenShift in many different ways. It can create pods from existing docker images, from Dockerfiles, and from raw source code using the Source-to-Image (S2I) process.

Run the `oc new-app -h` command to understand all the different options available for creating new applications on OpenShift.

The following command creates an application based on an image, `mysql`, from Docker Hub, with the label set to `db=mysql`:

```
[user@host ~]$ oc new-app mysql MYSQL_USER=user MYSQL_PASSWORD=pass
 MYSQL_DATABASE=testdb -l db=mysql
```

The following figure shows the Kubernetes and OpenShift resources created by the `oc new-app` command when the argument is a container image:

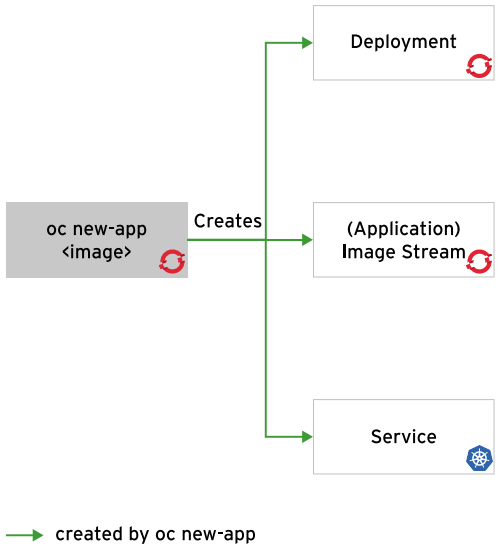


Figure 6.6: Resources created for a new application

The following command creates an application based on an image from a private Docker image registry:

```
oc new-app --docker-image=myregistry.com/mycompany/myapp --name=myapp
```

The following command creates an application based on source code stored in a Git repository:

```
oc new-app  https://github.com/openshift/ruby-hello-world  --name=ruby-hello
```

You will learn more about the Source-to-Image (S2I) process, its associated concepts, and more advanced ways to use `oc new-app` to build applications for OpenShift in the next section.

The following command creates an application based on an existing template:

```
$ oc new-app \
 --template=mysql-persistent \
 -p MYSQL_USER=user1 -p MYSQL_PASSWORD=mypa55 -p MYSQL_DATABASE=testdb \
 -p MYSQL_ROOT_PASSWORD=r00tpa55 -p VOLUME_CAPACITY=10Gi
...output omitted...
```

Note

You will learn more about templates on the next chapter.

Managing Persistent Storage

In addition to the specification of custom images, you can create persistent storage and attach it to your application. In this way you can make sure your data is not lost when deleting your pods. To list the `PersistentVolume` objects in a cluster, use the `oc get pv` command:

```
[admin@host ~]$ oc get pv
NAME     CAPACITY  ACCESS MODES  RECLAIM POLICY   STATUS      CLAIM   ...
pv0001   1Mi       RWO           Retain           Available           ...
pv0002   10Mi      RWX           Recycle          Available           ...
...output omitted...
```

To see the YAML definition for a given `PersistentVolume`, use the `oc get` command with the `-o yaml` option:

```
[admin@host ~]$ oc get pv pv0001 -o yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  creationTimestamp: ...value omitted...
  finalizers:
  - kubernetes.io/pv-protection
  labels:
    type: local
  name: pv0001
  resourceVersion: ...value omitted...
  selfLink: /api/v1/persistentvolumes/pv0001
  uid: ...value omitted...
spec:
```

```
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 1Mi
  hostPath:
    path: /data/pv0001
    type: ""
  persistentVolumeReclaimPolicy: Retain
status:
  phase: Available
```

To add more `PersistentVolume` objects to a cluster, use the `oc create` command:

```
[admin@host ~]$ oc create -f pv1001.yaml
```

Note

The above `pv1001.yaml` file must contain a persistent volume definition, similar in structure to the output of the `oc get pv` *`pv-name`* `-o yaml` command.

Requesting Persistent Volumes

When an application requires storage, you create a `PersistentVolumeClaim` (PVC) object to request a dedicated storage resource from the cluster pool. The following content from a file named `pvc.yaml` is an example definition for a PVC:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myapp
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

The PVC defines storage requirements for the application, such as capacity or throughput. To create the PVC, use the `oc create` command:

```
[admin@host ~]$ oc create -f pvc.yaml
```

After you create a PVC, OpenShift attempts to find an available `PersistentVolume` resource that satisfies the PVC's requirements. If OpenShift finds a match, it binds the PersistentVolume object to the PersistentVolumeClaim object. To list the PVCs in a project, use the `oc get pvc` command:

```
[admin@host ~]$ oc get pvc
NAME      STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
myapp     Bound    pv0001   1Gi        RWO                           6s
```

The output indicates whether a persistent volume is bound to the PVC, along with attributes of the PVC (such as capacity).

To use the persistent volume in an application pod, define a volume mount for a container that references the `PersistentVolumeClaim` object. The application pod definition below references a `PersistentVolumeClaim` object to define a volume mount for the application:

```
apiVersion: "v1"
kind: "Pod"
metadata:
  name: "myapp"
  labels:
    name: "myapp"
spec:
  containers:
    - name: "myapp"
      image: openshift/myapp
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/var/www/html"
          name: "pvol" (1)
  volumes:
    - name: "pvol" (2)
      persistentVolumeClaim:
        claimName: "myapp" (3)
```

(1) This section declares that the `pvol` volume mounts at `/var/www/html` in the container file system.

(2) This section defines the `pvol` volume.

(3) The `pvol` volume references the `myapp` PVC. If OpenShift associates an available persistent volume to the `myapp` PVC, then the `pvol` volume refers to this associated volume.

Managing OpenShift Resources at the Command Line

There are several essential commands used to manage OpenShift resources as described below.

Use the `oc get` command to retrieve information about resources in the cluster. Generally, this command outputs only the most important characteristics of the resources and omits more detailed information.

The `oc get` *`RESOURCE_TYPE`* command displays a summary of all resources of the specified type. The following illustrates example output of the `oc get pods` command.

```
NAME              READY   STATUS      RESTARTS  AGE
nginx-1-5r583     1/1     Running     0         1h
myapp-1-l44m7     1/1     Running     0         1h
```

oc get all

Use the `oc get all` command to retrieve a summary of the most important components of a cluster. This command iterates through the major resource types for the current project and prints out a summary of their information:

```
NAME       DOCKER REPO                                TAGS      UPDATED
is/nginx   172.30.1.1:5000/basic-kubernetes/nginx   latest    About an hour ago

NAME       REVISION   DESIRED   CURRENT   TRIGGERED BY
dc/nginx   1          1         1         config,image(nginx:latest)

NAME         DESIRED   CURRENT   READY     AGE
rc/nginx-1   1         1         1         1h

NAME        CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
svc/nginx   172.30.72.75   <none>        80/TCP,443/TCP   1h

NAME               READY     STATUS    RESTARTS   AGE
po/nginx-1-ypp8t   1/1       Running   0          1h
```

oc describe

If the summaries provided by `oc get` are insufficient, use the `oc describe RESOURCE_TYPE RESOURCE_NAME` command to retrieve additional information. Unlike the `oc get` command, there is no way to iterate through all the different resources by type. Although most major resources can be described, this functionality is not available across all resources. The following is an example output from describing a pod resource:

```
Name:       mysql-openshift-1-glqrp
Namespace:          mysql-openshift
Priority:           0
Node:               cluster-worker-1/172.25.250.52
Start Time:         Fri, 15 Feb 2019 02:14:34 +0000
Labels:             app=mysql-openshift
                    deployment=mysql-openshift-1
...output omitted...
Status:             Running
IP:                 10.129.0.85
```

oc get

The `oc get RESOURCE_TYPE RESOURCE_NAME` command can be used to export a resource definition. Typical use cases include creating a backup, or to aid in the modification of a definition. The `-o yaml` option prints out the object representation in YAML format, but this can be changed to JSON format by providing a `-o json` option.

oc create

This command creates resources from a resource definition. Typically, this is paired with the `oc get RESOURCE_TYPE RESOURCE_NAME -o yaml` command for editing definitions.

oc edit

This command allows the user to edit resources of a resource definition. By default, this command opens a `vi` buffer for editing the resource definition.

oc delete

The `oc delete RESOURCE_TYPE name` command removes a resource from an OpenShift cluster. Note that a fundamental understanding of the OpenShift architecture is needed here, because deleting managed resources such as pods results in new instances of those resources being automatically created. When a project is deleted, it deletes all of the resources and applications contained within it.

oc exec

The `oc exec CONTAINER_ID options` command executes commands inside a container. You can use this command to run interactive and noninteractive batch commands as part of a script.

Labelling Resources

When working with many resources in the same project, it is often useful to group those resources by application, environment, or some other criteria. To establish these groups, you define labels for the resources in your project. Labels are part of the `metadata` section of a resource, and are defined as key/value pairs, as shown in the following example:

```
apiVersion: v1
kind: Service
metadata:
...contents omitted...
  labels: app: nexus template: nexus-persistent-template
  name: nexus
...contents omitted...
```

Many `oc` subcommands support a `-l` option to process resources from a label specification. For the `oc get` command, the `-l` option acts as a selector to only retrieve objects that have a matching label:

```
$ oc get svc,deployments -l app=nexus
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
service/nexus   ClusterIP   172.30.29.218   <none>        8081/TCP   4h

NAME                                     REVISION   DESIRED   CURRENT   ...
deployment.apps.openshift.io/nexus   1          1         1         ...
```

Note

Although any label can appear in resources, both the `app` and `template` keys are common for labels. By convention, the `app` key indicates the application related to this resource. The `template` key labels all resources generated by the same template with the template's name.

When using templates to generate resources, labels are especially useful. A template resource has a `labels` section separated from the `metadata.labels` section. Labels defined in the `labels`

section do not apply to the template itself, but are added to every resource generated by the template:

```
apiVersion: template.openshift.io/v1
kind: Template
labels: app: nexus template: nexus-persistent-template
metadata:
...contents omitted...
labels: maintainer: redhat
  name: nexus-persistent
...contents omitted...
objects:
- apiVersion: v1
  kind: Service
  metadata:
    name: nexus
labels: version: 1
...contents omitted...
```

The previous example defines a template resource with a single label: `maintainer: redhat`. The template generates a service resource with three labels: `app: nexus`, `template: nexus-persistent-template`, and `version: 1`.

References

Additional information about pods and services is available in the *Pods and Services* section of the OpenShift Container Platform documentation:
Architecture
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/architecture/index

Additional information about creating images is available in the OpenShift Container Platform documentation:
Creating Images
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/images/index

Labels and label selectors details are available in *Working with Kubernetes Objects* section for the Kubernetes documentation:
Labels and Selectors
https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

Guided Exercise

Deploying a Database Server on OpenShift

In this exercise, you will create and deploy a MySQL database pod on OpenShift using the `oc new-app` command.

Outcomes

You should be able to create and deploy a MySQL database pod on OpenShift.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running Red Hat OpenShift (RHOCP) cluster. See *Guided Exercise: Deploying an Application to Red Hat OpenShift Container Platform* for more information about using the developer sandbox RHOCP cluster.
- The `oc` command.

Instructions

1. Log in with the `oc` utility.

1.1. Log in to the RHOCP web console.

1.2. In the RHOCP web console, click your username at the top right, and then click **Copy Login Command**. This will open a new tab and prompt you for your username and password.

1.3. Once authenticated, click **Display Token** and copy the provided `oc login` command.

1.4. Log in to your RHOCP account by using the copied command and notice the default project:

```
$ oc login --token=yourtoken --server=https://api.region.prod.nextcle.com:6443
Logged into "https://api.region.prod.nextcle.com:6443" as "youruser" using the
 token provided.
...output omitted...
Using project "youruser-dev".
```

2. Create a new application from the `mysql-persistent` template using the `oc new-app` command.

This image requires that you use the `-p` option to set the `MYSQL_USER`, `MYSQL_PASSWORD`, `MYSQL_DATABASE`, `MYSQL_ROOT_PASSWORD` and `VOLUME_CAPACITY` environment variables.

Use the `--template` option with the `oc new-app` command to specify a template with persistent storage so that OpenShift does not try and pull the image from the internet:

```
[student@workstation ~]$ oc new-app \
 --template=mysql-persistent \
 -p MYSQL_USER=user1 -p MYSQL_PASSWORD=mypa55 -p MYSQL_DATABASE=testdb \
 -p MYSQL_ROOT_PASSWORD=r00tpa55 -p VOLUME_CAPACITY=10Gi
--> Deploying template "openshift/mysql-persistent" to project
 ${RHT_OCP4_DEV_USER}-mysql-openshift
...output omitted...
--> Creating resources ...
    secret "mysql" created
    service "mysql" created
    persistentvolumeclaim "mysql" created
    deploymentconfig.apps.openshift.io "mysql" created
--> Success
    Application is not exposed. You can expose services to the outside world by
 executing one or more of the commands below:
     'oc expose service/mysql'
    Run 'oc status' to view your app.
```

3. Verify that the MySQL pod was created successfully and view the details about the pod and its service.

   3.1. Run the `oc status` command to view the status of the new application and verify that the deployment of the MySQL image was successful:

```
[student@workstation ~]$ oc status
In project youruser-dev on server ...

svc/mysql - 172.30.151.91:3306
...output omitted...
deployment #1 deployed 6 minutes ago - 1 pod
```

   3.2. List the pods in this project to verify that the MySQL pod is ready and running:

```
[student@workstation ~]$ oc get pods
NAME                     READY   STATUS      RESTARTS   AGE
mysql-1-5vfn4    1/1     Running     0          109s
```

Note

Notice the name of the running pod. You need this information to be able to log in to the MySQL database server later.

   3.3. Use the `oc describe` command to view more details about the pod:

```
[student@workstation ~]$ oc describe pod mysql-1-5vfn4
Name:         mysql-1-5vfn4
Namespace:    youruser-dev
Priority:     0
Node:         master01/192.168.50.10
Start Time:   Mon, 29 Mar 2021 16:42:13 -0400
Labels:       deployment=mysql-1
```

```
...output omitted...
Status:       Running
IP:           10.10.0.34
...output omitted...
```

3.4. List the services in this project and verify that the service to access the MySQL pod was created:

```
[student@workstation ~]$ oc get svc
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
mysql           ClusterIP   172.30.151.91   <none>        3306/TCP   10m
```

3.5. Retrieve the details of the `mysql` service using the `oc describe` command and note that the service type is `ClusterIP` by default:

```
[student@workstation ~]$ oc describe service mysql
Name:              mysql
Namespace:         ${RHT_OCP4_DEV_USER}-mysql-openshift
Labels:            app=mysql-persistent
                   app.kubernetes.io/component=mysql-persistent
                   app.kubernetes.io/instance=mysql-persistent
     template=mysql-persistent-template
Annotations:       openshift.io/generated-by: OpenShiftNewApp
...output omitted...
Selector:          name=mysql
Type:              ClusterIP
IP:                172.30.151.91
Port:              3306-tcp  3306/TCP
TargetPort:        3306/TCP
Endpoints:         10.10.0.34:3306
Session Affinity:  None
Events:            <none>
```

3.6. List the persistent storage claims in this project:

```
[student@workstation ~]$ oc get pvc
NAME    STATUS   VOLUME                                     CAPACITY   ...
 STORAGECLASS
mysql   Bound    pvc-e9bf0b1f-47df-4500-afb6-77e826f76c15   10Gi       ...
 standard
```

3.7. Retrieve the details of the `mysql` pvc using the `oc describe` command:

```
[student@workstation ~]$ oc describe pvc/mysql
Name:              mysql
Namespace:         ${RHT_OCP4_DEV_USER}-mysql-openshift
StorageClass:      standard
Status:            Bound
Volume:            pvc-e9bf0b1f-47df-4500-afb6-77e826f76c15
Labels:            app=mysql-persistent
                   app.kubernetes.io/component=mysql-persistent
                   app.kubernetes.io/instance=mysql-persistent
```

```
     template=mysql-persistent-template
Annotations:       openshift.io/generated-by: OpenShiftNewApp
...output omitted...
Capacity:          10Gi
Access Modes:      RWO
VolumeMode:        Filesystem
Mounted By:        mysql-1-5vfn4
```

4. Verify that the MySQL database was created successfully.

   4.1. Get the MySQL pod name:

```
[student@workstation ~]$ oc get pods
NAME             READY   STATUS      RESTARTS   AGE
mysql-1-5vfn4    1/1     Running     0          18m
mysql-1-deploy   0/1     Completed   0          18m
```

   4.2. Issue the `show databases;` SQL query to the MySQL pod:

```
[student@workstation ~]$ oc exec mysql-1-5vfn4 -- mysql -uuser1 -pmypa55 \
  --protocol tcp -h localhost -e 'show databases;'
Database
information_schema
testdb
```

5. Clean up.

   5.1. Remove RHOCP resources with the `app=mysql-persistent` label.

```
[student@workstation ~]$ oc delete all,secrets,pvc -l app=mysql-persistent
replicationcontroller "mysql-1" deleted
service "mysql" deleted
deploymentconfig.apps.openshift.io "mysql" deleted
secret "mysql" deleted
persistentvolumeclaim "mysql" deleted
```

Finish

This concludes the exercise.

Creating Routes

Objectives

After completing this section, students should be able to expose services using OpenShift routes.

Working with Routes

Services allow for network access between pods inside an OpenShift instance, and routes allow for network access to pods from users and applications outside the OpenShift instance.

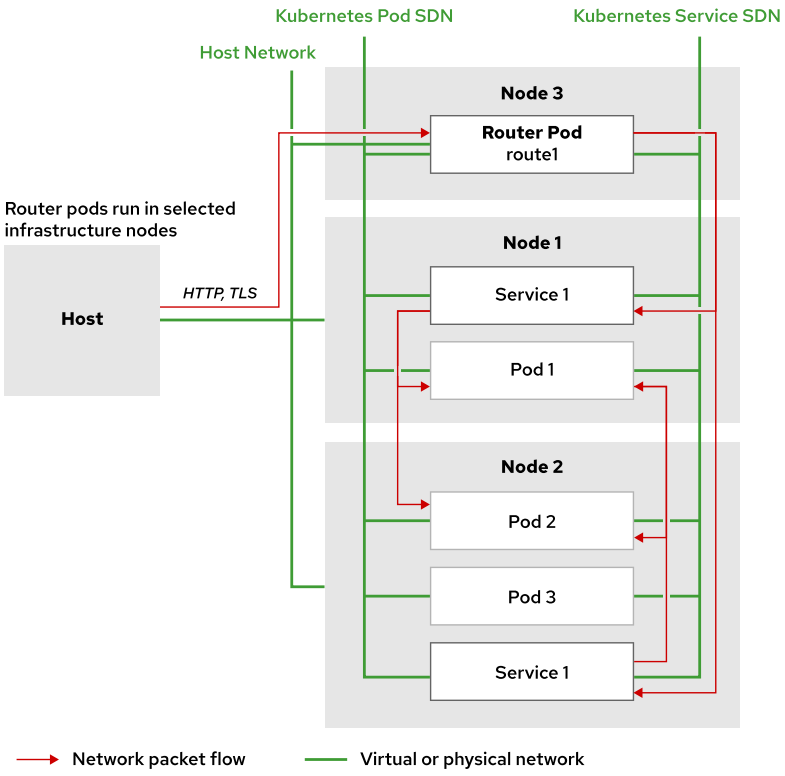


Figure 6.7: OpenShift routes and Kubernetes services

A route connects a public-facing IP address and DNS host name to an internal-facing service IP. It uses the service resource to find the endpoints; that is, the ports exposed by the service.

OpenShift routes are implemented by a cluster-wide router service, which runs as a containerized application in the OpenShift cluster. OpenShift scales and replicates router pods like any other OpenShift application.

Note

In practice, to improve performance and reduce latency, the OpenShift router connects directly to the pods using the internal pod software-defined network (SDN).

The router service uses *HAProxy* as the default implementation.

An important consideration for OpenShift administrators is that the public DNS host names configured for routes need to point to the public-facing IP addresses of the nodes running the router. Router pods, unlike regular application pods, bind to their nodes' public IP addresses instead of to the internal pod SDN.

The following example shows a minimal route defined using JSON syntax:

```
{
    "apiVersion": "v1",
    "kind": "Route",
    "metadata": {
        "name": "quoteapp"
    },
    "spec": {
        "host": "quoteapp.apps.example.com",
        "to": {
            "kind": "Service",
            "name": "quoteapp"
        }
    }
}
```

The `apiVersion`, `kind`, and `metadata` attributes follow standard Kubernetes resource definition rules. The `Route` value for `kind` shows that this is a route resource, and the `metadata.name` attribute gives this particular route the identifier `quoteapp`.

As with pods and services, the main part is the `spec` attribute, which is an object containing the following attributes:

- `host` is a string containing the FQDN associated with the route. DNS must resolve this FQDN to the IP address of the OpenShift router. The details to modify DNS configuration are outside the scope of this course.
- `to` is an object stating the resource this route points to. In this case, the route points to an OpenShift Service with the `name` set to `quoteapp`.

Note

Names of different resource types do not collide. It is perfectly legal to have a route named `quoteapp` that points to a service also named `quoteapp`.

Important

Unlike services, which use selectors to link to pod resources containing specific labels, a route links directly to the service resource name.

Creating Routes

Use the `oc create` command to create route resources, just like any other OpenShift resource. You must provide a JSON or YAML resource definition file, which defines the route, to the `oc create` command.

The `oc new-app` command does not create a route resource when building a pod from container images, Dockerfiles, or application source code. After all, `oc new-app` does not know if the pod is intended to be accessible from outside the OpenShift instance or not.

Another way to create a route is to use the `oc expose service` command, passing a service resource name as the input. The `--name` option can be used to control the name of the route resource. For example:

```
$ oc expose service quotedb --name quote
```

By default, routes created by `oc expose` generate DNS names of the form: `route-name-project-name.default-domain`

Where:

- *route-name* is the name assigned to the route. If no explicit name is set, OpenShift assigns the route the same name as the originating resource (for example, the service name).
- *project-name* is the name of the project containing the resource.
- *default-domain* is configured on the OpenShift Control Plane and corresponds to the wildcard DNS domain listed as a prerequisite for installing OpenShift.

For example, creating a route named `quote` in project named `test` from an OpenShift instance where the wildcard domain is `cloudapps.example.com` results in the FQDN `quote-test.cloudapps.example.com`.

Note

The DNS server that hosts the wildcard domain knows nothing about route host names. It merely resolves any name to the configured IP addresses. Only the OpenShift router knows about route host names, treating each one as an HTTP virtual host. The OpenShift router blocks invalid wildcard domain host names that do not correspond to any route and returns an HTTP 404 error.

Leveraging the Default Routing Service

The default routing service is implemented as an `HAProxy` pod. Router pods, containers, and their configuration can be inspected just like any other resource in an OpenShift cluster:

```
$ oc get pod --all-namespaces | grep router
openshift-ingress  router-default-746b5cfb65-f6sdm 1/1    Running  1         4d
```

Note that you can query information on the default router using the associated label as shown here.

By default, router is deployed in `openshift-ingress` project. Use `oc describe pod` command to get the routing configuration details:

```
$ oc describe pod router-default-746b5cfb65-f6sdm
Name:               router-default-746b5cfb65-f6sdm
Namespace:          openshift-ingress
...output omitted...
Containers:
  router:
...output omitted...
    Environment:
      STATS_PORT:                 1936
      ROUTER_SERVICE_NAMESPACE:   openshift-ingress
      DEFAULT_CERTIFICATE_DIR:    /etc/pki/tls/private
      ROUTER_SERVICE_NAME:        default
      ROUTER_CANONICAL_HOSTNAME:  apps.cluster.lab.example.com
...output omitted...
```

The subdomain, or default domain to be used in all default routes, takes its value from the `ROUTER_CANONICAL_HOSTNAME` entry.

References

Additional information about the architecture of routes in OpenShift is available in the *Architecture* and *Developer Guide* sections of the **OpenShift Container Platform documentation.**
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/

Guided Exercise

Exposing a Service as a Route

In this exercise, you will create, build, and deploy an application on an OpenShift cluster and expose its service as a route.

Outcomes

You should be able to expose a service as a route for a deployed OpenShift application.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running Red Hat OpenShift (RHOCP) cluster. See *Guided Exercise: Deploying an Application to Red Hat OpenShift Container Platform* for more information about using the developer sandbox RHOCP cluster.
- The `oc` command.

Instructions

1. Log in with the `oc` utility.

   1.1. Log in to the RHOCP web console.

   1.2. In the RHOCP web console, click your username at the top right, and then click **Copy Login Command**. This will open a new tab and prompt you for your username and password.

   1.3. Once authenticated, click **Display Token** and copy the provided `oc login` command.

   1.4. Log in to your RHOCP account by using the copied command and notice the default project:

```
$ oc login --token=yourtoken --server=https://api.region.prod.nextcle.com:6443
Logged into "https://api.region.prod.nextcle.com:6443" as "youruser" using the
 token provided.
...output omitted...
Using project "youruser-dev".
```

2. Create a new PHP application using the `quay.io/redhattraining/php-hello-dockerfile` image.

   2.1. Use the `oc new-app` command to create the PHP application.

Important

The following example uses a backslash (\) to indicate that the second line is a continuation of the first line. If you wish to ignore the backslash, you can type the entire command in one line.

```
[student@workstation ~]$ oc new-app \
 --docker-image=quay.io/redhattraining/php-hello-dockerfile \
 --name php-helloworld
--> Found container image 4b696cc (2 years old) from quay.io for "quay.io/
redhattraining/php-hello-dockerfile"
...output omitted...
--> Creating resources ...
...output omitted...
--> Success
    Application is not exposed. You can expose services to the outside world by
 executing one or more of the commands below:
     'oc expose service/php-helloworld'
    Run 'oc status' to view your app.
```

2.2. Wait until the application finishes deploying by monitoring the progress with the `oc get pods -w` command:

```
[student@workstation ~]$ oc get pods -w
NAME                              READY   STATUS    RESTARTS   AGE
php-helloworld-74bb86f6cb-zt6wl   1/1     Running   0          5s
^C
```

Your exact output may differ in names, status, timing, and order. The container in `Running` status with a random suffix (`74bb86f6cb-zt6wl` in the example) contains the application and shows it is up and running. Alternatively, monitor the deployment logs with the `oc logs -f php-helloworld-74bb86f6cb-zt6wl` command. Press Ctrl + C to exit the command if necessary.

```
[student@workstation ~]$ oc logs -f php-helloworld-74bb86f6cb-zt6wl
AH00558: httpd: Could not reliably determine the server's fully qualified domain
 name, using 10.129.5.124. Set the 'ServerName' directive globally to suppress
 this message
[09-Aug-2021 22:09:45] NOTICE: [pool www] 'user' directive is ignored when FPM is
 not running as root
[09-Aug-2021 22:09:45] NOTICE: [pool www] 'group' directive is ignored when FPM is
 not running as root
^C
```

Your exact output may differ.

2.3. Review the service for this application using the `oc describe` command:

```
[student@workstation ~]$ oc describe svc/php-helloworld
Name:              php-helloworld
Namespace:         youruser-dev
Labels:            app=php-helloworld
                   app.kubernetes.io/component=php-helloworld
                   app.kubernetes.io/instance=php-helloworld
Annotations:       openshift.io/generated-by: OpenShiftNewApp
Selector:          deployment=php-helloworld
Type:              ClusterIP
IP:                172.30.100.236
```

```
Port:              8080-tcp  8080/TCP
TargetPort:        8080/TCP
Endpoints:         10.129.5.124:8080
Session Affinity:  None
Events:            <none>
```

The IP address and namespace displayed in the output of the command may differ.

3. Expose the service, which creates a route. Use the default name and fully qualified domain name (FQDN) for the route:

```
[student@workstation ~]$ oc expose svc/php-helloworld
route.route.openshift.io/php-helloworld exposed
```

3.1. Verify the host name of the route:

```
[student@workstation ~]$ oc describe route
Name:             php-helloworld
Namespace:        youruser-dev
Created:          16 seconds ago
Labels:           app=php-helloworld
                  app.kubernetes.io/component=php-helloworld
                  app.kubernetes.io/instance=php-helloworld
Annotations:      openshift.io/host.generated=true
Requested Host:   php-helloworld-youruser-dev.apps.sandbox-
m2.ll9k.p1.openshiftapps.com
                    exposed on router default (host router-default.apps.sandbox-
m2.ll9k.p1.openshiftapps.com) 16 seconds ago
Path:             <none>
TLS Termination:  <none>
Insecure Policy:  <none>
Endpoint Port:    8080-tcp

Service:    php-helloworld
Weight:     100 (100%)
Endpoints:  10.129.5.124:8080
```

4. Access the service in a web browser to verify that the service and route are working.

For example, to use Firefox, execute:

```
[student@workstation ~]$ firefox \
 php-helloworld-youruser-dev-route.apps.na46-stage2.dev.nextcle.com
```

Notice the FQDN is comprised of the application name and project name by default. The remainder of the FQDN, the subdomain, is defined when OpenShift is installed.

5. Replace this route with a route named xyz.

5.1. Delete the current route:

```
[student@workstation ~]$ oc delete route/php-helloworld
route.route.openshift.io "php-helloworld" deleted
```

Note

Deleting the route is optional. You can have multiple routes for the same service, provided they have different names.

5.2. Create a route for the service with a name of *`youruser`*`-xyz`. Replace *youruser* with your user.

```
[student@workstation ~]$ oc expose svc/php-helloworld \
 --name=yoususer-xyz
route.route.openshift.io/yoususer-xyz exposed
```

5.3. Verify the host name of the route:

```
[student@workstation ~]$ oc describe route
Name:             youruser-xyz
Namespace:        youruser-dev
Created:          23 seconds ago
Labels:           app=php-helloworld
                  app.kubernetes.io/component=php-helloworld
                  app.kubernetes.io/instance=php-helloworld
Annotations:      openshift.io/host.generated=true
Requested Host:   youruser-xyz-youruser-dev.apps.sandbox-
m2.ll9k.p1.openshiftapps.com
                    exposed on router default (host router-default.apps.sandbox-
m2.ll9k.p1.openshiftapps.com) 22 seconds ago
Path:             <none>
TLS Termination:  <none>
Insecure Policy:  <none>
Endpoint Port:    8080-tcp

Service:    php-helloworld
Weight:     100 (100%)
Endpoints:  10.129.5.124:8080
```

Your exact output may differ. Note the new FQDN that was generated based on the new route name. Both the route name and the project name contain your user name, hence it appears twice in the route FQDN.

5.4. Access the route host in a web browser.

For example:

```
[student@workstation ~]$ firefox \
 youruser-xyz-youruser-dev.apps.sandbox-m2.ll9k.p1.openshiftapps.com
```

6. Clean up.

6.1. Remove RHOCP resources with the `app=php-helloworld` label.

```
[student@workstation ~]$ oc delete all -l app=php-helloworld
service "php-helloworld" deleted
deployment.apps "php-helloworld" deleted
imagestream.image.openshift.io "php-helloworld" deleted
route.route.openshift.io "youruser-xyz" deleted
```

Finish

This concludes the guided exercise.

Creating Applications with Source-to-Image

Objectives

After completing this section, students should be able to deploy an application using the Source-to-Image (S2I) facility of OpenShift Container Platform.

The Source-to-Image (S2I) Process

Source-to-Image (S2I) is a tool that makes it easy to build container images from application source code. This tool takes an application's source code from a Git repository, injects the source code into a base container based on the language and framework desired, and produces a new container image that runs the assembled application.

This figure shows the resources created by the `oc new-app` command when the argument is an application source code repository. Notice that S2I also creates a Deployment and all its dependent resources:

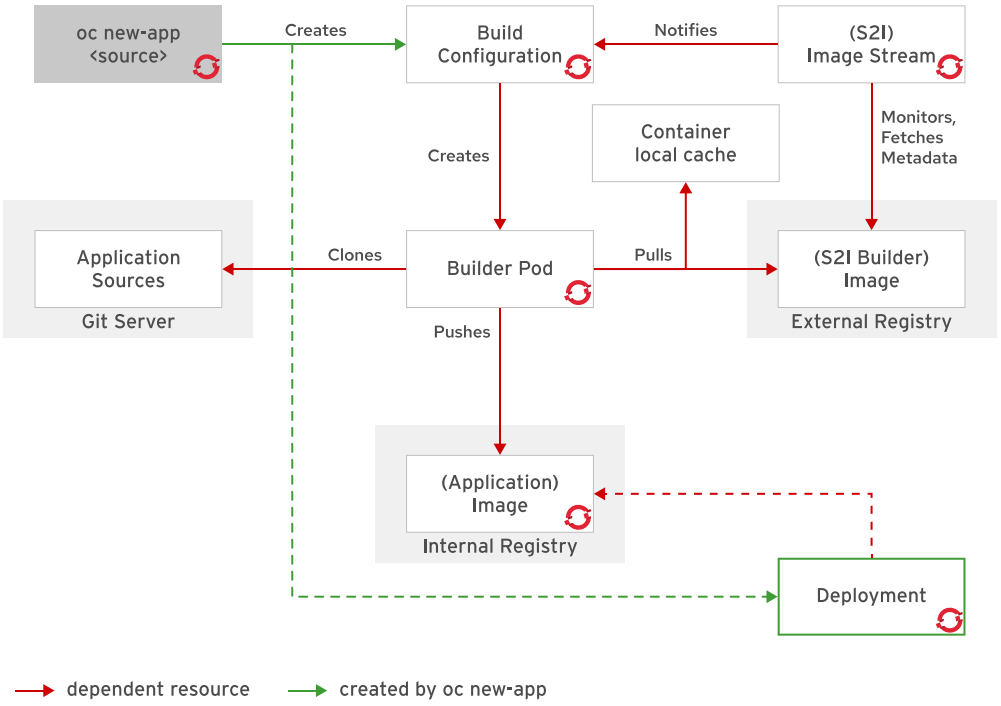


Figure 6.8: Deployment and dependent resources

S2I is the primary strategy used for building applications in OpenShift Container Platform. The main reasons for using source builds are:

- User efficiency: Developers do not need to understand Dockerfiles and operating system commands such as `yum install`. They work using their standard programming language tools.

- Patching: S2I allows for rebuilding all the applications consistently if a base image needs a patch due to a security issue. For example, if a security issue is found in a PHP base image, then updating this image with security patches updates all applications that use this image as a base.

- Speed: With S2I, the assembly process can perform a large number of complex operations without creating a new layer at each step, resulting in faster builds.

- Ecosystem: S2I encourages a shared ecosystem of images where base images and scripts can be customized and reused across multiple types of applications.

Describing Image Streams

OpenShift deploys new versions of user applications into pods quickly. To create a new application, in addition to the application source code, a base image (the S2I builder image) is required. If either of these two components gets updated, OpenShift creates a new container image. Pods created using the older container image are replaced by pods using the new image.

Even though it is evident that the container image needs to be updated when application code changes, it may not be evident that the deployed pods also need to be updated should the builder image change.

The image stream resource is a configuration that names specific container images associated with image stream tags, an alias for these container images. OpenShift builds applications against an image stream. The OpenShift installer populates several image streams by default during installation. To determine available image streams, use the `oc get` command, as follows:

```
$ oc get is -n openshift
NAME           IMAGE REPOSITORY                          TAGS
cli            ...svc:5000/openshift/cli                 latest
dotnet         ...svc:5000/openshift/dotnet              2.1,...,3.1-el7,latest
dotnet-runtime ...svc:5000/openshift/dotnet-runtime      2.1,...,3.1-el7,latest
httpd          ...svc:5000/openshift/httpd               2.4,2.4-el7,2.4-el8,latest
jenkins        ...svc:5000/openshift/jenkins             2,latest
mariadb        ...svc:5000/openshift/mariadb             10.3,10.3-el7,10.3-el8,latest
mongodb        ...svc:5000/openshift/mongodb             3.6,latest
mysql          ...svc:5000/openshift/mysql               8.0,8.0-el7,8.0-el8,latest
nginx          ...svc:5000/openshift/nginx               1.10,1.12,1.8,latest
nodejs         ...svc:5000/openshift/nodejs              10,...,12-ubi7,12-ubi8
perl           ...svc:5000/openshift/perl                5.26,...,5.30,5.30-el7
php            ...svc:5000/openshift/php                 7.2-ubi8,...,7.3-ubi8,latest
postgresql     ...svc:5000/openshift/postgresql          10,10-el7,...,12-el7,12-el8
python         ...svc:5000/openshift/python              2.7,2.7-ubi7,...,3.6-ubi8,3.8
redis          ...svc:5000/openshift/redis               5,5-el7,5-el8,latest
ruby           ...svc:5000/openshift/ruby                2.5,2.5-ubi7,...,2.6,2.6-ubi7
...
```

Note

Your OpenShift instance may have more or fewer image streams depending on local additions and OpenShift point releases.

OpenShift detects when an image stream changes and takes action based on that change. If a security issue arises in the `rhel8/nodejs-10` image, it can be updated in the image repository, and OpenShift can automatically trigger a new build of the application code.

It is likely that an organization chooses several supported base S2I images from Red Hat, but may also create their own base images.

Building an Application with S2I and the CLI

Building an application with S2I can be accomplished using the OpenShift CLI.

An application can be created using the S2I process with the `oc new-app` command from the CLI:

```
$ oc new-app php~http://my.git.server.com/my-app \ 1
 --name=myapp 2
```

1. The image stream used in the process appears to the left of the tilde (~) character.

   The URL after the tilde indicates the location of the source code's Git repository.

2. Sets the application name.

Note

Instead of using the tilde character, you can set the image stream by using the `-i` option or `--image-stream` for the full version.

```
$ oc new-app -i php  http://services.lab.example.com/app  --name=myapp
```

The `oc new-app` command allows creating applications using source code from a local or remote Git repository. If only a source repository is specified, `oc new-app` tries to identify the correct image stream to use for building the application. In addition to application code, S2I can also identify and process Dockerfiles to create a new image.

The following example creates an application using the Git repository in the current directory:

```
$ oc new-app .
```

Important

When using a local Git repository, the repository must have a remote origin that points to a URL accessible by the OpenShift instance.

It is also possible to create an application using a remote Git repository and a context subdirectory:

```
$ oc new-app  https://github.com/openshift/sti-ruby.git  \
  --context-dir=2.0/test/puma-test-app
```

Finally, it is possible to create an application using a remote Git repository with a specific branch reference:

```
$ oc new-app  https://github.com/openshift/ruby-hello-world.git#beta4
```

If an image stream is not specified in the command, `new-app` attempts to determine which language builder to use based on the presence of certain files in the root of the repository:

Language	Files
Ruby	`Rakefile` `Gemfile` `config.ru`
Java EE	`pom.xml`
Node.js	`app.json` `package.json`
PHP	`index.php` `composer.json`
Python	`requirements.txt` `config.py`
Perl	`index.pl` `cpanfile`

After a language is detected, the `new-app` command searches for image stream tags that have support for the detected language, or an image stream that matches the name of the detected language.

Create a JSON resource definition file by using the `-o json` parameter and output redirection:

```
$ oc -o json new-app php~http://services.lab.example.com/app \
 --name=myapp > s2i.json
```

This JSON definition file creates a list of resources. The first resource is the image stream:

```
...output omitted...
{
    "kind": "ImageStream", (1)
    "apiVersion": "image.openshift.io/v1",
    "metadata": {
        "name": "myapp", (2)
        "creationTimestamp": null
        "labels": {
                    "app": "myapp"
                },
                "annotations": {
                    "openshift.io/generated-by": "OpenShiftNewApp"
                }
    },
    "spec": {
        "lookupPolicy": {
            "local": false
        }
    },
    "status": {
        "dockerImageRepository": ""
    }
},
...output omitted...
```

(1) Define a resource type of image stream.

2 Name the image stream `myapp`.

The build configuration (`bc`) is responsible for defining input parameters and triggers that are executed to transform the source code into a runnable image. The `BuildConfig` (BC) is the second resource, and the following example provides an overview of the parameters used by OpenShift to create a runnable image.

```
...output omitted...
{
    "kind": "BuildConfig", 1
    "apiVersion": "build.openshift.io/v1",
    "metadata": {
        "name": "myapp", 2
        "creationTimestamp": null,
        "labels": {
            "app": "myapp"
        },
        "annotations": {
            "openshift.io/generated-by": "OpenShiftNewApp"
        }
    },
    "spec": {
        "triggers": [
            {
                "type": "GitHub",
                "github": {
                    "secret": "S5_4BZpPabM6KrIuPBvI"
                }
            },
            {
                "type": "Generic",
                "generic": {
                    "secret": "3q8K8JNDoRzhjoz1KgMz"
                }
            },
            {
                "type": "ConfigChange"
            },
            {
                "type": "ImageChange",
                "imageChange": {}
            }
        ],
        "source": {
            "type": "Git",
            "git": {
                "uri": "http://services.lab.example.com/app" 3
            }
        },
        "strategy": {
            "type": "Source", 4
            "sourceStrategy": {
                "from": {
                    "kind": "ImageStreamTag",
                    "namespace": "openshift",
```

```
                    "name": "php:7.3" 5
                }
            }
        },
        "output": {
            "to": {
                "kind": "ImageStreamTag",
                "name": "myapp:latest" 6
            }
        },
        "resources": {},
        "postCommit": {},
        "nodeSelector": null
    },
    "status": {
        "lastVersion": 0
    }
},
...output omitted...
```

1. Define a resource type of `BuildConfig`.
2. Name the `BuildConfig` `myapp`.
3. Define the address to the source code Git repository.
4. Define the strategy to use S2I.
5. Define the builder image as the `php:7.3` image stream.
6. Name the output image stream `myapp:latest`.

The third resource is the deployment object that is responsible for customizing the deployment process in OpenShift. It may include parameters and triggers that are necessary to create new container instances, and are translated into a replication controller from Kubernetes. Some of the features provided by `Deployment` objects are:

- User customizable strategies to transition from the existing deployments to new deployments.
- Have as many active replica set as wanted and possible.
- Replication scaling depends of the sizes of old and new replica sets.

```
...output omitted...
        {
            "kind": "Deployment", 1
            "apiVersion": "apps/v1",
            "metadata": {
                "name": "myapp", 2
                "creationTimestamp": null,
                "labels": {
                    "app": "myapp",
                    "app.kubernetes.io/component": "myapp",
                    "app.kubernetes.io/instance": "myapp"
                },
                "annotations": {
```

```
                    "image.openshift.io/triggers": "[{\"from\":{\"kind\":
\"ImageStreamTag\",\"name\":\"myapp:l
atest\"},\"fieldPath\":\"spec.template.spec.containers[?(@.name==\\\"myapp\\
\")].image\"}]", (3)
                    "openshift.io/generated-by": "OpenShiftNewApp"
                }
            },
            "spec": {
                "replicas": 1,
                "selector": {
                    "matchLabels": {
                        "deployment": "myapp"
                    }
                },
                "template": {
                    "metadata": {
                        "creationTimestamp": null,
                        "labels": {
                            "deployment": "myapp" (4)
                        },
                        "annotations": {
                            "openshift.io/generated-by": "OpenShiftNewApp"
                        }
                    },
                    "spec": {
                        "containers": [
                            {
                                "name": "myapp", (5)
                                "image": " ",
                                "ports": [ (6)
                                    {
                                        "containerPort": 8080,
                                        "protocol": "TCP"
                                    },
                                    {
                                        "containerPort": 8443,
                                        "protocol": "TCP"
                                    }
                                ],
                                "resources": {}
                            }
                        ]
                    }
                },
                "strategy": {}
            },
            "status": {}
        },
...output omitted...
```

(1) Define a resource type of `Deployment`.

(2) Name the `Deployment` `myapp`.

3. An image change trigger causes the creation of a new deployment each time a new version of the `myapp:latest` image is available in the repository.

4. A configuration change trigger causes a new deployment to be created any time the replication controller template changes.

5. Defines the container image to deploy: `myapp:latest`.

6. Specifies the container ports. A configuration change trigger causes a new deployment to be created any time the replication controller template changes.

The last item is the service, already covered in previous chapters:

```
...output omitted...
{
    "kind": "Service",
    "apiVersion": "v1",
    "metadata": {
        "name": "myapp",
        "creationTimestamp": null,
        "labels": {
            "app": "myapp"
            "app.kubernetes.io/component": "myapp",
            "app.kubernetes.io/instance": "myapp"
        },
        "annotations": {
            "openshift.io/generated-by": "OpenShiftNewApp"
        }
    },
    "spec": {
        "ports": [
            {
                "name": "8080-tcp",
                "protocol": "TCP",
                "port": 8080,
                "targetPort": 8080
            },
            {
                "name": "8443-tcp",
                "protocol": "TCP",
                "port": 8443,
                "targetPort": 8443
            }
        ],
        "selector": {
            "deployment": "myapp"
        }
    },
    "status": {
        "loadBalancer": {}
    }
}
```

Note

By default, the `oc new-app` command does not create a route. You can create a route after creating the application. However, a route is automatically created when using the web console because it uses a template.

After creating a new application, the build process starts. Use the `oc get builds` command to see a list of application builds:

```
$ oc get builds
NAME               TYPE     FROM          STATUS    STARTED          DURATION
php-helloworld-1   Source   Git@9e17db8   Running   13 seconds ago
```

OpenShift allows viewing the build logs. The following command shows the last few lines of the build log:

```
$ oc logs build/myapp-1
```

Important

If the build is not Running yet, or OpenShift has not deployed the `s2i-build` pod yet, the above command throws an error. Just wait a few moments and retry it.

Trigger a new build with the `oc start-build build_config_name` command:

```
$ oc get buildconfig
NAME           TYPE      FROM      LATEST
myapp          Source    Git       1
```

```
$ oc start-build myapp
build "myapp-2" started
```

Relationship Between Build and Deployment

The `BuildConfig` pod is responsible for creating the images in OpenShift and pushing them to the internal container registry. Any source code or content update typically requires a new build to guarantee the image is updated.

The `Deployment` pod is responsible for deploying pods to OpenShift. The outcome of a `Deployment` pod execution is the creation of pods with the images deployed in the internal container registry. Any existing running pod may be destroyed, depending on how the `Deployment` resource is set.

The `BuildConfig` and `Deployment` resources do not interact directly. The `BuildConfig` resource creates or updates a container image. The `Deployment` reacts to this new image or updated image event and creates pods from the container image.

References

Source-to-Image (S2I) Build
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/builds/build-strategies#builds-strategy-s2i-build_understanding-image-builds

S2I GitHub repository
https://github.com/openshift/source-to-image

Guided Exercise

Creating a Containerized Application with Source-to-Image

In this exercise, you will build an application from source code and deploy the application to an OpenShift cluster.

Outcomes

You should be able to:

- Build an application from source code using the OpenShift command-line interface.
- Verify the successful deployment of the application using the OpenShift command-line interface.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running Red Hat OpenShift (RHOCP) cluster. See *Guided Exercise: Deploying an Application to Red Hat OpenShift Container Platform* for more information about using the developer sandbox RHOCP cluster.
- The source code for the `php-helloworld` application in the `DO101x-apps` Git repository on your local system.
- The `oc` command.

Instructions

1. Create and push a new branch named `s2i` to use during this exercise.

1.1. Launch the Visual Studio Code (VS Code) editor. In the Explorer view (**View** > **Explorer**), open the `DO101x-apps` folder in the `My Projects` workspace. The source code for the `version` application is in the `version` directory.

1.2. Ensure that the `DO101x-apps` repository uses the `main` branch. If you were working with another branch for a different exercise, click the current branch and then select `main` in the `Select a ref to checkout` window to switch to the `main` branch.

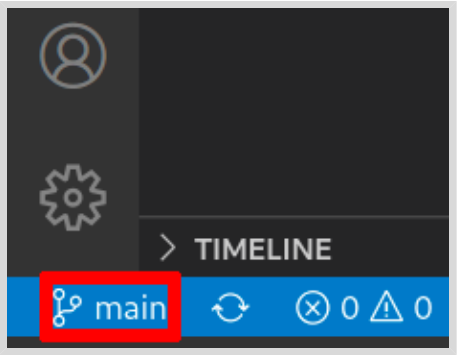


Figure 6.9: Select the main branch

Warning
Each exercise uses a unique branch. Always change to the `main` branch before you create a new branch. The new branch must use the `main` branch as the base.

1.3. Click the **main** branch in VS Code.
Select `Create new branch...` from the list of options.

1.4. At the prompt, enter `s2i` for the branch name.

1.5. Push the `s2i` branch to your `DO101x-apps` GitHub repository.
Click the **Publish Changes** cloud icon next to the `update-app` branch name to push your local branch to your remote Git repository.

2. Review the PHP source code of the application, inside the the `php-helloworld` directory.
Open the `index.php` file:

```
<?php
print "Hello, World! php version is " . PHP_VERSION . "\n";
?>
```

The application implements a simple response which returns the PHP version it is running.

3. Log in with the `oc` utility.

3.1. Log in to the RHOCP web console.

3.2. In the RHOCP web console, click your username at the top right, and then click **Copy Login Command**. This will open a new tab and prompt you for your username and password.

3.3. Once authenticated, click **Display Token** and copy the provided `oc login` command.

3.4. Log in to your RHOCP account by using the copied command and notice the default project:

```
$ oc login --token=yourtoken --server=https://api.region.prod.nextcle.com:6443
Logged into "https://api.region.prod.nextcle.com:6443" as "youruser" using the
 token provided.
...output omitted...
Using project "youruser-dev".
```

4. Create a new PHP application by using the Source-to-Image deployment method. Use the `php-helloworld` directory in the `s2i` branch you created in the previous step in your fork of the `DO101x-apps` Git repository.

   4.1. Use the `oc new-app` command to create the PHP application.

Important

The following example uses the number sign (#) to select a specific branch from the git repository, in this case the `s2i` branch created in the previous step.

```
[student@workstation ~]$ oc new-app php:latest --name=php-helloworld \
  https://github.com/YOUR_GITHUB_USERNAME/DO101x-apps#s2i \
 --context-dir php-helloworld
--> Found image 576cf8b (7 months old) in image stream "openshift/php" under tag
 "latest" for "php:latest"
...output omitted...
```

   4.2. Wait for the build to complete and the application to deploy. Verify that the build process starts with the `oc get pods` command.

```
[student@workstation ~]$ oc get pods
NAME                     READY   STATUS    RESTARTS   AGE
php-helloworld-1-build   1/1     Running   0          5s
```

   4.3. Examine the logs for this build. Use the build pod name for this build, `php-helloworld-1-build`.

```
[student@workstation ~]$ oc logs -f php-helloworld-1-build
...output omitted...

Writing manifest to image destination
Storing signatures
Generating dockerfile with builder image image-registry.openshift-image-...
php@sha256:3206...37b4
Adding transient rw bind mount for /run/secrets/rhsm
STEP 1: FROM image-registry.openshift-image-registry.svc:5000...

...output omitted...

STEP 8: RUN /usr/libexec/s2i/assemble

...output omitted...
```

```
Pushing image .../php-helloworld:latest ...
Getting image source signatures

...output omitted...

Writing manifest to image destination
Storing signatures
Successfully pushed .../php-helloworld@sha256:3f1c...c454
Push successful
Cloning "https://github.com/YOUR_GITHUB_USER/DO101x-apps" ...
...output omitted...
```

Notice that the clone of the Git repository is the first step of the build. Next, the Source-to-Image process built a new image called `s2i/php-helloworld:latest`. The last step in the build process is to push this image to the OpenShift private registry.

4.4. Review the `Deployment` for this application:

```
[student@workstation ~]$ oc describe deployment/php-helloworld
Name:                   php-helloworld
Namespace:              youruser-dev
CreationTimestamp:      Tue, 30 Mar 2021 12:54:59 -0400
Labels:                 app=php-helloworld
                        app.kubernetes.io/component=php-helloworld
                        app.kubernetes.io/instance=php-helloworld
Annotations:            deployment.kubernetes.io/revision: 2
                        image.openshift.io/triggers:
                          [{"from":{"kind":"ImageStreamTag","name":"php-
helloworld:latest"},"fieldPath":"spec.template.spec.containers[?(@.name==\"php-
helloworld\")...
                        openshift.io/generated-by: OpenShiftNewApp
Selector:               deployment=php-helloworld
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0
 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:       deployment=php-helloworld
  Annotations:  openshift.io/generated-by: OpenShiftNewApp
  Containers:
   php-helloworld:
    Ports:        8080/TCP, 8443/TCP
    Host Ports:   0/TCP, 0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
```

```
OldReplicaSets:  <none>
NewReplicaSet:   php-helloworld-6f5d4c47ff (1/1 replicas created)
...output omitted...
```

4.5. Add a route to test the application:

```
[student@workstation ~]$ oc expose service php-helloworld
route.route.openshift.io/php-helloworld exposed
```

4.6. Find the URL associated with the new route:

```
[student@workstation ~]$ oc get route -o jsonpath="{..spec.host}{'\n'}"
php-helloworld-youruser-dev.apps.sandbox-m2.ll9k.p1.openshiftapps.com
```

4.7. Test the application by opening the URL you obtained in the previous step in a web browser, for example:

```
[student@workstation ~]$ firefox \
 php-helloworld-youruser-dev.apps.sandbox-m2.ll9k.p1.openshiftapps.com
```

5. Explore starting application builds by changing the application in its Git repository and executing the proper commands to start a new Source-to-Image build.

5.1. Edit the `index.php` file as shown below:

```
<?php
print "Hello, World! php version is " . PHP_VERSION . "\n";
print "A change is a coming!\n";
?>
```

Save the file.

5.2. Commit your changes locally, and then push the new commit to your GitHub repository.

Switch to the `Source Control` view (**View** > **SCM**). Locate the entry for the `index.php` file under the `CHANGES` heading for the `DO101x-apps` directory.

Click the plus (+) button to add the `index.php` file changes to your next commit.

5.3. Add a commit message of `updated app to v2` in the message prompt, and then click the check mark button to commit the staged changes.

5.4. Click the **Views and more actions** > **push** to publish the changes to the remote repository.

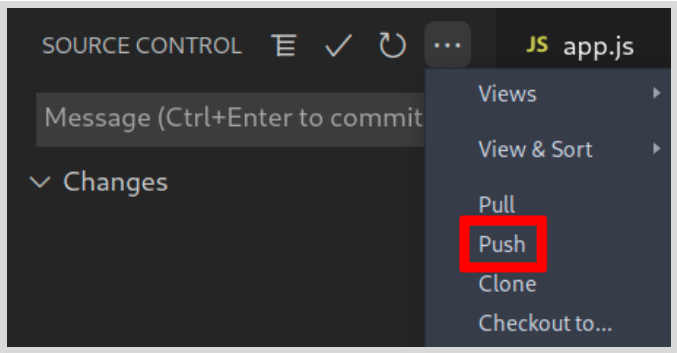


Figure 6.10: Push changes to remote repository

5.5. Start a new Source-to-Image build process.

```
[student@workstation php-helloworld]$ oc start-build php-helloworld
build.build.openshift.io/php-helloworld-2 started
```

5.6. Wait for the new build to finish:

```
[student@workstation php-helloworld]$ oc logs php-helloworld-2-build -f
...output omitted...

Successfully pushed .../php-helloworld:latest@sha256:74e757a4c0edaeda497dab7...
Push successful
```

Note
Logs may take a while to be available after the build starts. If the previous command fails, wait and try again.

5.7. After the second build has completed use the `oc get pods` command to verify that the new version of the application is running.

```
[student@workstation php-helloworld]$ oc get pods
NAME                      READY   STATUS      RESTARTS   AGE
php-helloworld-1-build    0/1     Completed   0          11m
php-helloworld-2-build    0/1     Completed   0          45s
php-helloworld-2-wq9wz    1/1     Running     0          13s
```

5.8. Test that the application serves the new content:

```
[student@workstation php-helloworld]$ firefox \
 php-helloworld-youruser-dev.apps.sandbox-m2.ll9k.p1.openshiftapps.com
```

6. Clean up.

6.1. Remove RHOCP resources with the `app=php-helloworld` label.

```
[student@workstation ~]$ oc delete all -l app=php-helloworld
service "php-helloworld" deleted
deployment.apps "php-helloworld" deleted
buildconfig.build.openshift.io "php-helloworld" deleted
imagestream.image.openshift.io "php-helloworld" deleted
route.route.openshift.io "php-helloworld" deleted
```

Finish

This concludes the guided exercise.

Summary

- OpenShift Container Platform stores definitions of each OpenShift or Kubernetes resource instance as an object in the cluster's distributed database service, `etcd`. Common resource types are: `Pod`, `Persistent Volume` (PV), `Persistent Volume Claim` (PVC), `Service` (SVC), `Route`, `Deployment`, `DeploymentConfig` and `Build Configuration` (BC).
- Use the OpenShift command-line client `oc` to:
  - Create, change, and delete projects.
  - Create application resources inside a project.
  - Delete, inspect, edit, and export resources inside a project.
  - Check logs from application pods, deployments, and build operations.
- The `oc new-app` command can create application pods in many different ways: from an existing container image hosted on an image registry, from Dockerfiles, and from source code using the Source-to-Image (S2I) process.
- Source-to-Image (S2I) is a tool that makes it easy to build a container image from application source code. This tool retrieves source code from a Git repository, injects the source code into a selected container image based on a specific language or technology, and produces a new container image that runs the assembled application.
- A `Route` connects a public-facing IP address and DNS host name to an internal-facing service IP. While services allow for network access between pods inside an OpenShift instance, routes allow for network access to pods from users and applications outside the OpenShift instance.
- You can create, build, deploy and monitor applications using the OpenShift web console.